

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

是时候，将深度学习应用到实践了！
全面提升深度学习的应用能力！占领未来科技的至高点！

深度学习

——Caffe之经典模型详解与实战

乐毅 王斌 编著



作者简介

乐毅:

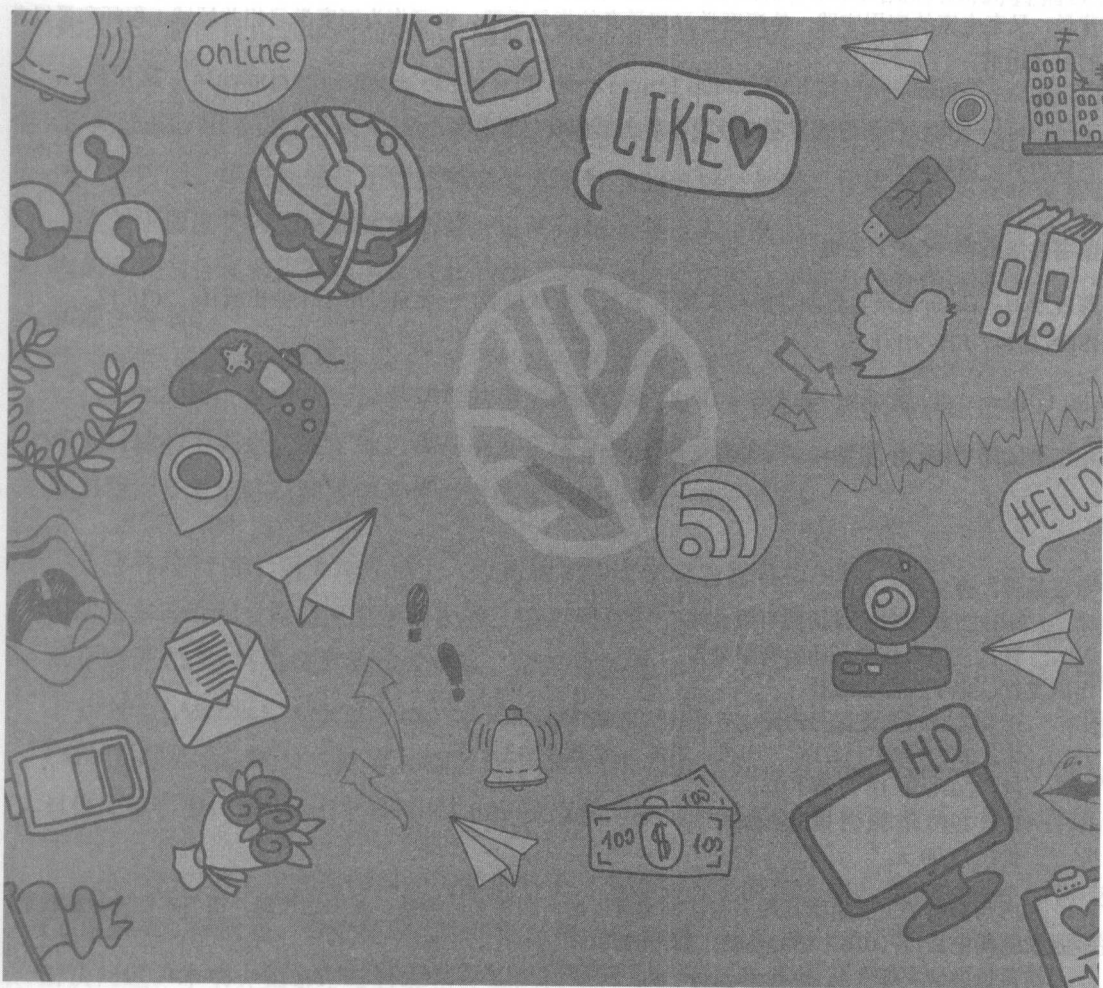
计算机专业硕士，现任职于某数据通信公司，高级系统工程师。负责公司深度学习技术领域的应用及相关项目，对深度学习及大数据深度挖掘具有浓厚的兴趣。擅长 Caffe 等深度学习框架及网络模型应用。

王斌:

通信与信息系统硕士，现任职于某数据通信公司，高级系统工程师。多年致力于深度学习技术的前沿研究与应用，对 Caffe 等深度学习框架在图像识别领域有深刻理解，承担公司多项与机器学习相关的研究工作。

——Caffe之经典模型详解与实战

乐毅 王斌 编著



電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书首先介绍了深度学习相关的理论和主流的深度学习框架,然后从Caffe深度学习框架为切入点,介绍了Caffe的安装、配置、编译和接口等运行环境,剖析Caffe网络模型的构成要素和常用的层类型和Solver方法。通过LeNet网络模型的Mnist手写实例介绍其样本训练和识别过程,进一步详细解读了AlexNet、VGGNet、GoogLeNet、Siamese和SqueezeNet网络模型,并给出了这些模型基于Caffe的训练实战方法。然后,本书解读了利用深度学习进行目标定位的经典网络模型:FCN、R-CNN、Fast-RCNN、Faster-RCNN和SSD,并进行目标定位Caffe实战。本书的最后,从著名的Kaggle网站引入了两个经典的实战项目,并进行了有针对性的原始数据分析、网络模型设计和Caffe训练策略实践,以求带给读者从问题提出到利用Caffe求解的完整工程经历,从而使读者能尽快掌握Caffe框架的使用技巧和实战经验。

针对Caffe和深度学习领域的初学者,本书是一本不可多得的参考资料。本书的内容既有易懂的理论背景,又有丰富的应用实践,是深度学习初学者的指导手册,也可作为深度学习相关领域工程师和爱好者的参考用书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

深度学习: Caffe之经典模型详解与实战 / 乐毅, 王斌编著. —北京: 电子工业出版社, 2016.12
ISBN 978-7-121-30118-6

I. ①深… II. ①乐… ②王… III. ①学习系统 IV. ①TP273

中国版本图书馆CIP数据核字(2016)第247611号

责任编辑: 孙学瑛

印 刷: 三河市鑫金马印装有限公司

装 订: 三河市鑫金马印装有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编100036

开 本: 787×980 1/16 印张: 21.5 字数: 333千字

版 次: 2016年12月第1版

印 次: 2016年12月第1次印刷

定 价: 79.00元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: (010) 51260888-819 faq@phei.com.cn。

前言

2016年3月，Google开发的一款人工智能程序阿尔法围棋（AlphaGo）对战世界围棋冠军、职业九段选手李世石，以4:1的总比分获胜。众多媒体和网络新闻纷纷直播或转载此次人工智能应用领域内的盛况。随后，Google在《Nature》杂志发表了关于AlphaGo算法的论文“Mastering the game of Go with deep neural networks and tree search”。此论文提到了AlphaGo用3,000万棋局训练深度神经网络的方法，展现了深度学习异常强大的学习能力。一时间，国内外掀起了研究和学习人工智能的热潮。然而，很多读者朋友希望能找到一本关于深度学习应用领域的书籍，目前市场上关于人工智能、机器学习或深度学习领域内的各类书目很多，遗憾的是这些书籍大多是理论性质的，少有系统介绍深度学习的应用实践参考书。

与此同时，笔者认为深度学习的应用能力会成为一个爆发性需求的知识技能，也会是未来科技的至高点。鉴于此，我与朋友王斌从去年就计划编写一本关于深度学习的应用实践书籍，希望能够对深度学习爱好者或初学者提供一些帮助。

全书共17章，第1章介绍了人工智能和深度学习的背景和现状；第2章介绍了深度学习的基本理论和主流的深度学习框架；第3章介绍了Caffe的安装、配置和运行环境；第4章介绍了Caffe网络模型的构成要素、常用的层类型和Solver方法；第5~10章详细解读了LeNet、AlexNet、GoogLeNet、VGGNet、Siamese和SqueezeNet目标分类模型，并附上Caffe实战训练；第11~15章详细解读了FCN、R-CNN、Fast-RCNN、Faster-RCNN和SSD目标定位模型；第16~17章利用Caffe深度学习框架求解Kaggle网站的两个经典项目。

本书在内容上对深度学习相关的机器学习理论只作了简单介绍，更多的放在如何应用Caffe解决实际问题，并把使用当中可能出现的问题也一一列举出来，帮助读者分析原因、解决问题。本书介绍了十多种非常经典的网络模型，学习这些模型可以帮助读者很好地理解和应用Caffe框架和工具。当然，读者并无必要对这些网络模型一一阅读，也可根据自

IV ▶ 前言

身情况选择对自己有实际帮助的案例进行分析学习。

由于深度学习技术发展迅速,各种知识和应用工具变化很快,Github 上许多开源的项目也在不断更新和修正。笔者才疏学浅,理解有限,加之编写时间也较仓促,书中难免有错谬之处,敬请广大读者朋友批评指正,不胜感激。

乐毅

2016年11月

目 录

第1章 绪论	1
1.1 引言	1
1.2 人工智能的发展历程	2
1.3 机器学习及相关技术	4
1.3.1 学习形式分类	4
1.3.2 学习方法分类	5
1.3.3 机器学习的相关技术	7
1.4 国内外研究现状	8
1.4.1 国外研究现状	8
1.4.2 国内研究现状	9
第2章 深度学习	11
2.1 神经网络模型	11
2.1.1 人脑视觉机理	11
2.1.2 生物神经元	13
2.1.3 人工神经网络	15
2.2 BP神经网络	18
2.2.1 BP神经元	18
2.2.2 BP神经网络构成	19
2.2.3 正向传播	21
2.2.4 反向传播	21
2.3 卷积神经网络	24
2.3.1 卷积神经网络的历史	25
2.3.2 卷积神经网络的网络结构	26
2.3.3 局部感知	27
2.3.4 参数共享	28
2.3.5 多卷积核	28

2.3.6 池化 (Pooling)	29
2.4 深度学习框架	30
2.4.1 Caffe	30
2.4.2 Torch	31
2.4.3 Keras	32
2.4.4 MXNet	32
2.4.5 TensorFlow	33
2.4.6 CNTK	33
2.4.7 Theano	34
第3章 Caffe 简介及其安装配置	36
3.1 Caffe 是什么	36
3.1.1 Caffe 的特点	38
3.1.2 Caffe 的架构	38
3.2 Caffe 的安装环境	39
3.2.1 Caffe 的硬件环境	39
3.2.2 Caffe 的软件环境	43
3.2.3 Caffe 的依赖库	44
3.2.4 Caffe 开发环境的安装	46
3.3 Caffe 接口	52
3.3.1 Caffe Python 接口	52
3.3.2 Caffe MATLAB 接口	55
3.3.3 Caffe 命令行接口	56
第4章 Caffe 网络定义	58
4.1 Caffe 模型要素	58
4.1.1 网络模型	58
4.1.2 参数配置	62
4.2 Google Protobuf 结构化数据	63
4.3 Caffe 数据库	65
4.3.1 LevelDB	65
4.3.2 LMDB	66
4.3.3 HDF5	66
4.4 Caffe Net	66

4.5	Caffe Blob	68
4.6	Caffe Layer	70
4.6.1	Data Layers	71
4.6.2	Convolution Layers	75
4.6.3	Pooling Layers	76
4.6.4	InnerProduct Layers	77
4.6.5	ReLU Layers	78
4.6.6	Sigmoid Layers	79
4.6.7	LRN Layers	79
4.6.8	Dropout Layers	80
4.6.9	SoftmaxWithLoss Layers	80
4.6.10	Softmax Layers	81
4.6.11	Accuracy Layers	81
4.7	Caffe Solver	82
	Solver 方法	83
第 5 章	LeNet 模型	88
5.1	LeNet 模型简介	88
5.2	LeNet 模型解读	89
5.3	Caffe 环境 LeNet 模型	91
5.3.1	mnist 实例详解	91
5.3.2	mnist 手写测试	103
5.3.3	mnist 样本字库的图片转换	106
第 6 章	AlexNet 模型	107
6.1	AlexNet 模型介绍	107
6.2	AlexNet 模型解读	108
6.3	AlexNet 模型特点	111
6.4	Caffe 环境 AlexNet 模型训练	112
6.4.1	数据准备	112
6.4.2	其他支持文件	113
6.4.3	图片预处理	113
6.4.4	ImageNet 数据集介绍	113
6.4.5	ImageNet 图片介绍	115

6.4.6	ImageNet 模型训练	115
6.4.7	Caffe 的 AlexNet 模型与论文的不同	124
6.4.8	ImageNet 模型测试	124
第 7 章	GoogLeNet 模型	126
7.1	GoogLeNet 模型简介	126
7.1.1	背景和动机	127
7.1.2	Inception 结构	127
7.2	GoogLeNet 模型解读	129
7.2.1	GoogLeNet 模型结构	129
7.2.2	GoogLeNet 模型特点	134
7.3	GoogLeNet 模型的 Caffe 实现	135
第 8 章	VGGNet 模型	146
8.1	VGGNet 网络模型	146
8.1.1	VGGNet 模型介绍	146
8.1.2	VGGNet 模型特点	147
8.1.3	VGGNet 模型解读	147
8.2	VGGNet 网络训练	149
8.2.1	VGGNet 训练参数设置	149
8.2.2	Multi-Scale 训练	149
8.2.3	测试	150
8.2.4	部署	150
8.3	VGGNet 模型分类实验	150
8.3.1	Single-scale 对比	150
8.3.2	Multi-scale 对比	151
8.3.3	模型融合	152
8.4	VGGNet 网络结构	153
第 9 章	Siamese 模型	158
9.1	Siamese 网络模型	159
9.1.1	Siamese 模型原理	159
9.1.2	Siamese 模型实现	160
9.2	Siamese 网络训练	165

9.2.1	数据准备	165
9.2.2	生成 side	165
9.2.3	对比损失函数	166
9.2.4	定义 solver	166
9.2.5	网络训练	166
第 10 章 SqueezeNet 模型		168
10.1	SqueezeNet 网络模型	168
10.1.1	SqueezeNet 模型原理	168
10.1.2	Fire Module	169
10.1.3	SqueezeNet 模型结构	170
10.1.4	SqueezeNet 模型特点	171
10.2	SqueezeNet 网络实现	172
第 11 章 FCN 模型		177
11.1	FCN 模型简介	177
11.2	FCN 的特点和使用场景	178
11.3	Caffe FCN 解读	179
11.3.1	FCN 模型训练准备	180
11.3.1	FCN 模型训练	183
第 12 章 R-CNN 模型		196
12.1	R-CNN 模型简介	196
12.2	R-CNN 的特点和使用场景	197
12.3	Caffe R-CNN 解读	198
12.3.1	R-CNN 模型训练准备	198
12.3.2	R-CNN 模型训练	201
第 13 章 Fast-RCNN 模型		217
13.1	Fast-RCNN 模型简介	217
13.2	Fast-RCNN 的特点和使用场景	218
13.3	Caffe Fast-RCNN 解读	220
13.3.1	Fast-RCNN 模型训练准备	220
13.3.2	Fast-RCNN 模型训练	222

第 14 章 Faster-RCNN 模型	239
14.1 Faster-RCNN 模型简介	239
14.2 Faster-RCNN 的特点和使用场景	241
14.3 Caffe Faster-RCNN 解读	242
14.3.1 Faster-RCNN 模型训练准备	242
14.3.2 Faster-RCNN 模型训练	244
第 15 章 SSD 模型	264
15.1 SSD 模型简介	264
15.2 SSD 的特点和使用场景	266
15.3 Caffe SSD 解读	267
15.3.1 SSD 模型训练准备	267
15.3.2 SSD 模型训练	268
第 16 章 Kaggle 项目实践：人脸特征检测	290
16.1 项目简介	290
16.2 赛题和数据	291
16.3 Caffe 训练和测试数据库	293
16.3.1 数据库生成	293
16.3.2 网络对比	295
16.3.3 网络一	296
16.3.4 网络二	300
16.3.5 Python 人脸特征预测程序	306
第 17 章 Kaggle 项目实践：猫狗分类检测	311
17.1 项目简介	311
17.2 赛题和数据	312
17.3 Caffe 训练和测试数据库	312
17.3.1 数据库生成	312
17.3.2 Caffe 实现	316
17.3.3 CatdogNet 训练	328
17.3.4 CatdogNet 模型验证	332

1

第 1 章 绪论

深度学习是机器学习研究中的一个新领域，其动机在于建立能模拟人脑进行分析学习的神经网络，它模仿人脑的机制来解释数据，例如图像、声音和文本。深度学习算法基于人工神经网络，是当前机器学习领域最前沿和热门的课题之一。本章将介绍机器学习和深度学习的发展历程、相关技术和国内外研究现状。

1.1 引言

人工神经网络 (Artificial Neural Networks, ANNs)，简称为神经网络 (NNs) 或连接模型 (Connection Model)，它是一种模仿动物神经网络行为特征，进行分布式并行信息处理的算法数学模型。神经网络是计算智能和机器学习研究领域内非常活跃的分支之一。深度学习 (Deep Learning) 算法是近年来在人工神经网络领域的一项重大突破。深度学习通过组合低层特征形成更加抽象的高层特征，来发现数据的分布式特征。在深度学习中，我们并不告诉计算机如何解决问题，相反，计算机自己能够从观测数据中学习，然后通过学习结果，自行解决问题。

从 2006 年开始发展起来的深度学习算法经过多年的快速发展已经在计算机视觉、语音识别和自然语言处理中的许多重要问题上有杰出的表现。由于深度学习的先进性，已经使

得机器智能开始加速走进人类的生活。在深度学习算法发展的过程中，有一些深度学习框架随之发展起来并广受欢迎。本书介绍的 Caffe 是一款优秀的神经网络深度学习框架，在深度学习领域广受欢迎。本书的目的是帮助读者掌握深度学习的核心概念，并深入浅出地介绍 Caffe 的内容和基于 Caffe 的深度学习模型。

1.2 人工智能的发展历程

人工智能的研究从 1956 年正式开始，这一年在达特茅斯大学召开的会议上正式使用了“人工智能”（Artificial Intelligence, AI）这个术语。从计算机应用系统的角度出发，人工智能是研究如何制造智能机器或智能系统，来模拟人类智能活动的能力，以延伸人类智能的科学。人工智能理论的发展历程至今为止可分为三个阶段。

第一阶段：初始阶段

1950 年，著名的图灵测试诞生，按照艾伦·图灵的定义：如果一台机器能够与人类展开对话（通过电传设备）而不能被辨别出其机器身份，那么称这台机器具有智能。同年，图灵还预言了创造出具有真正智能的机器的可能性。

1956 年，在达特茅斯学院举行的一次会议上，计算机科学家约翰·麦卡锡说服与会者接受“人工智能”一词作为本领域的名称。后来，这次会议也被大家看成人工智能正式诞生的标志。

人工智能概念首次提出后，相继出现了一批显著的成果，如机器定理证明、跳棋程序、LISP 表处理语言等。但由于消解法推理能力的有限，以及机器翻译等的失败，使人工智能走入了低谷。这一阶段的特点是：重视问题求解的方法，忽视知识的重要性。

第二阶段：发展阶段

20 世纪 60 年代末到 20 世纪 70 年代，专家系统出现。DENDRAL 化学质谱分析系统、MYCIN 疾病诊断和治疗系统、PROSPECTOR 探矿系统、Hearsay-II 语音理解系统等专家

系统的研究和开发，将人工智能引向了实用化。

20 世纪 80 年代，Astrom 发表了论文，这是第一篇直接将人工智能的专家系统技术引入到控制系统的代表作，明确地提出了建立专家控制的新概念。与此同时，对于模糊理论的研究，以及其他智能理论的分支，都开始迅速白热化并展开研究，这些标志着智能控制已从研制开发阶段转向应用阶段。

20 世纪 80 年代末期，用于人工神经网络的反向传播算法（也叫 Back Propagation 算法或者 BP 算法）的发明，掀起了基于统计模型的机器学习热潮。这种基于统计的机器学习方法比起过去基于人工规则的系统，在很多方面显示出优越性。这个时候的人工神经网络，虽然也被称作多层感知机（Multi-layer Perceptron）^[1]，但实际上是一种只含有一层隐层节点的浅层模型。

20 世纪 90 年代，由于网络技术特别是 Internet 技术的发展，人工智能开始由单个智能主体研究转向基于网络环境下的分布式人工智能研究。不仅研究基于同一目标的分布式问题求解，而且研究多个智能主体的多目标问题求解，使人工智能进一步面向实用。里程碑似的成果包括在 1997 年 IBM 的深蓝战胜国际象棋世界冠军卡斯帕罗夫。这个时期，各种各样的浅层机器学习模型相继被提出，比如支撑向量机（Support Vector Machines, SVM）^[2]、Boosting^[3]、最大熵方法（例如 Logistic Regression, LR）等。这些模型的结构基本上可以看成带有一层隐层节点（如 SVM、Boosting）或没有隐层节点（如 LR）。

2000 年以来互联网的高速发展，对大数据的智能化分析和预测提出了巨大的需求，浅层学习模型在互联网应用上获得了巨大的成功。非常成功的应用包括搜索广告系统（例如 Google 的 AdWords、百度的凤巢系统）的广告点击率 CTR 预估、网页搜索排序（例如 Yahoo! 和微软的搜索引擎）、垃圾邮件过滤系统、基于内容的推荐系统等。

第三阶段：大数据+深度模型阶段

2006 年，加拿大多伦多大学教授、机器学习领域泰斗——Geoffrey Hinton 和他的学生 Ruslan Salakhutdinov 在顶尖学术刊物《科学》上发表了一篇文章，开启了深度学习在学术界和工业界的浪潮。这篇文章有两个主要的信息：一是多隐层的人工神经网络具有优异的

特征学习能力，学习得到的特征对数据有更本质的刻画，从而有利于可视化或分类；二是深度神经网络在训练上的难点，可以通过“逐层初始化”来有效克服。

自2006年以来，深度学习在学术界持续升温。目前深度学习的理论研究虽然还处于起步阶段，但在应用领域已显现出巨大能量。2011年以来，微软研究院和Google的语音识别研究人员先后采用深度神经网络（DNN）技术降低语音识别错误率达20%~30%，是语音识别领域十多年来最大的突破性进展。2012年，DNN技术在图像识别领域取得惊人的效果，在ImageNet评测上将错误率从26%降低到15%。同样在这一年，DNN还被应用于制药公司的DrugeActivity预测问题，并获得世界上的最好成绩。2015年，微软在ImageNet评测上的错误率已降低至3.57%^[4]，低于人眼判别的错误率（大约是5.1%）。2016年3月，Google的围棋软件AlphaGo对战世界围棋冠军、职业九段选手李世石，并以4:1的总比分获胜。

今天Google、Facebook、微软等知名的拥有大数据的高科技公司争相投入资源，占领深度学习的技术制高点。因为他们都看到了在大数据时代，更加复杂且更加强度的深度模型能深刻揭示海量数据里所承载的复杂而丰富的信息，并能够对未来或未知事件做更精准的预测。

深度学习带来了机器学习的新浪潮，推动“大数据+深度模型”时代的来临，并推动人工智能和人机交互等关键技术大步前进。

1.3 机器学习及相关技术

1.3.1 学习形式分类

1. 监督学习（Supervised Learning）

监督学习，也被称为有监督学习或有教师学习。监督学习是从标记的训练数据来推断一个功能的机器学习任务。训练数据包括一套训练示例。在监督学习中，每个实例都是由一个输入对象（通常为矢量）和一个期望的输出值（也称为监督信号）组成的。监督学习

从给定的训练数据集中学习一个函数，当新的数据到来时，可以根据这个函数预测结果。监督学习的训练集需要包括输入和输出，也可以说是特征和目标。训练集中的目标是由人标注的。常见的监督学习算法包括回归分析算法（regression）和统计分类算法（classification）。这一类学习主要应用于分类和预测。

2. 无监督学习（unsupervised learning）

无监督学习是从没有标记的训练数据中学习数据的信息或特征，无监督学习通过对没有标记的训练样本进行学习，来发现训练样本集中的结构性知识。这里，所有的标记是未知的，因此训练样本的歧义性高。聚类就是典型的无监督学习，因为给学习者提供的实例是未标记的，所以也没有错误或报酬信号来评估潜在的解决方案。

无监督学习和统计数据密度估计密切相关。除此之外，无监督学习还包括寻求、总结和解释数据的主要特点等诸多技术。在无监督学习中，使用的许多方法是基于数据挖掘方法。

1.3.2 学习方法分类

1. 经验性归纳学习

经验性归纳学习采用一些数据密集的经验方法（如版本空间法、ID3 法，定律发现方法）对例子进行归纳学习。其例子和学习结果一般都采用属性、谓词、关系等符号表示。它相当于基于学习策略分类中的归纳学习，但扣除连接学习、遗传算法、加强学习部分。

2. 分析学习

分析学习方法是从小数几个实例出发，运用领域知识进行分析。其主要特征为：

- 推理策略主要是演绎，而非归纳；
- 使用过去求解问题的经验（实例）指导新的问题求解或形成一定的问题求解规则；

- 分析学习的目标是改善系统的性能，而不是新的概念描述。分析学习包括应用解释学习、演绎学习、多级结构组块，以及宏操作学习等技术。

3. 类比学习

类比学习相当于基于学习策略分类中的类比学习。在这一类型的学习中比较引人注目的研究是通过与过去经历的具体事例作类比来学习，这样的学习被称为基于范例的学习(case_based learning)，或简称范例学习。

4. 遗传算法

遗传算法模拟生物繁殖的突变、交换和达尔文的自然选择（在每一生态环境中适者生存）。它把问题中的每一个个体编码为一个向量，向量的每一个元素被称为基因，并利用目标函数（相应于自然选择标准）对群体（个体的集合）中的每一个个体进行评价，根据评价值（适应度）对个体进行选择、交换、变异等遗传操作，从而得到新的群体。遗传算法适用于非常复杂和困难的环境，比如，带有大量噪声和无关数据、事物不断更新、问题目标不能明显和精确地定义，以及通过很长的执行过程才能确定当前行为的价值等。同神经网络一样，遗传算法的研究已经发展为人工智能的一个独立分支，其代表人物为霍勒德(J.H.Holland)。

5. 连接学习

典型的连接模型实现为人工神经网络，模型由神经元的一些简单计算单元及单元间的加权连接组成。

6. 增强学习

增强学习的特点是通过与环境的试探性(trial and error)交互来确定和优化动作的选择，以实现所谓的序列决策任务。在这种任务中，学习机制通过选择并执行动作，导致系统状态的变化，并有可能得到某种强化信号，从而实现与环境的交互。强化信号就是对系统行为的一种标量化的奖惩。增强学习的目标定义为寻找一个最优化的动作选择策略，即在任

一给定的状态下选择哪种动作的方法，使产生的动作序列可获得某种最优的结果。

1.3.3 机器学习的相关技术

1. BP (Back Propagation) 神经网络

BP 神经网络是 1986 年由 Rumelhart 和 McClelland 为首的科学家小组提出的，是一种按误差逆传播算法训练的多层前馈网络，是目前应用最广泛的神经网络模型之一。BP 网络能学习和存储大量的输入-输出模式映射关系，而无须事前揭示描述这种映射关系的数学方程。它的学习规则是使用梯度下降法，通过反向传播来不断调整网络的权值和阈值，使网络的误差平方和最小。BP 神经网络模型拓扑结构包括输入层 (Input layer)、隐含层 (Hidden layer) 和输出层 (Output layer)。

2. 随机森林 (Random Forests)

在机器学习中，随机森林是一个包含多个决策树的分类器，并且随机森林的输出类别是由森林中的每棵树输出类别的统计量综合决定的。这个方法结合了 Breiman 的“Bootstrap aggregating”思想和 Ho 的“Random Subspace Method”思想来建造决策树的集合。

3. 支持向量机 (Support Vector Machine, SVM)

在机器学习中，支持向量 (SVM) 是一种监督学习模型，可以分析数据、识别模式，用于分类和回归分析。支持向量机的一个关键概念就是核函数，可以使用所谓的核技巧，把输入隐含映射成高维特征空间，有效地进行线性分类或非线性分类。

4. 深度学习 (Deeping Learning)

深度学习的概念由 Hinton 等人于 2006 年提出。采用深度学习算法的典型神经网络采用大量的卷积层，利用空间相对关系来减少参数数目以提高训练性能。基于深度学习的 Caffe 学习框架及其经典模型的介绍是本书的重点。

1.4 国内外研究现状

1.4.1 国外研究现状

2006年,加拿大多伦多大学教授、机器学习领域的泰斗 Geoffrey Hinton^[5]和他的学生 Ruslan Salakhutdinov 在《科学》上发表了一篇基于神经网络深度学习理念的突破性文章,开启了深度学习在学术界和工业界的浪潮。

2010年,美国国防部先进研究项目局首次资助深度学习,参与方包括斯坦福大学、纽约大学和 NEC 美国研究院等机构。

2011年,斯坦福人工智能实验室主任吴恩达领导 Google 的科学家们,用 16000 台电脑模拟了一个人脑神经网络,并向这个网络展示了 1000 万段随机从 YouTube 上选取的视频,看看它能学会什么。结果在完全没有外界干涉的条件下,它自己识别出了猫脸。

2011年,微软语音识别采用深度学习技术降低语音识别错误率达 20 ~ 30%,是该领域十多年来最大的突破性进展。

2012 是深度学习研究和应用爆发的一年,深度学习被应用于著名生物制药公司默克的分子药性预测问题,从各类分子中学习发现那些可能成为药物分子,并获得世界上最好效果。

2012年,Hinton 和他的学生 Alex Krizhevsky 为了回应别人对于 Deep Learning 的质疑而将其用于 ImageNet (图像识别目前最大的数据库)上,最终取得了非常惊人的结果,其结果相对原来的最佳纪录有巨大的提升(前 5 项错误率由 25%降低为 17%)。Alex 在比赛中所用的神经模型在业界被称为 AlexNet^[6]。深度学习从此风靡世界。

2012年12月,微软亚洲研究院展示了中英即时口译系统,错误率仅为 7%,而且发音十分顺畅。

2013年,欧洲委员会发起模仿人脑的超级计算机项目,计划历时 10 年投入 16 亿美元,由全球 80 个机构的超过 200 名研究人员共同参与,希望在理解人类大脑工作方式上取得重

大进展，并推动更多能力强大的新型计算机的研发。

2013年6月18日，微软宣布已经研发出一种新型语音识别技术，可提供“接近即时”的语音至文本的转换服务，比当时最先进的语音识别技术快两倍，同时，准确率提高了15%。

2014年，GoogLeNet^[7]和VGG-Net^[8]获得ImageNet 2014计算机识别竞赛的冠亚军，这两类模型结构有一个共同特点是go deeper。GoogLeNet用的参数比ImageNet 2012计算机识别挑战赛的冠军AlexNet少12倍，但准确率更高。最终的Top-5错误率在验证集和测试集上都是6.67%，获得了第一名。

2015年微软亚洲研究院视觉计算组在ImageNet 2015计算机识别挑战赛中凭借深层神经网络技术的最新突破，以绝对优势获得图像分类、图像定位和图像检测全部三个主要项目的冠军。该研究团队使用了“残差学习”原理来指导神经网络结构的设计。“残差学习”最重要的突破在于重构了学习的过程，并重新定义了深层神经网络中的信息流，很好地解决了此前深层神经网络层级与准确度之间的矛盾。前5项错误率是3.57%，而此前同样的实验中，人眼辨识的错误率大概为5.1%。

2016年3月，Google开发的一款围棋人工智能程序阿尔法围棋（AlphaGo）对战世界围棋冠军、职业九段选手李世石，并以4:1的总比分获胜。

1.4.2 国内研究现状

关于深度学习在国内的发展情况，百度、华为、阿里巴巴、腾讯等中国著名的信息技术和互联网企业近年来也纷纷高举深度学习的旗帜，投入到深度学习研究的大军中去。

2012年，华为在中国香港成立诺亚方舟实验室，主要从事人工智能学习——数据挖掘研究。

2013年01月19日，百度宣布成立深度学习研究院Institute of Deep Learning(简称IDL)。之后宣布了百度大脑计划。

2014年06月22日，腾讯在ICML展示了Deep Learning Platform深度学习平台。

2014年09月09日，京东挂牌成立了京东深度神经网络实验室（JD DNN Lab）。

2014年11月03日, 阿里巴巴组建团队成立 IDST (Institute of Data Science & Tech)。

2016年5月23日, 发改委印发《“互联网+”人工智能三年行动实施方案》, 明确指出要推进计算机视觉、智能语音处理、生物特征识别、自然语言理解、智能决策控制等技术, 推进人工智能在家居、汽车、无人系统、安防等方面的应用, 提升人工智能集群式创新创业能力。这已经是国务院在人工智能领域的第二次发文。以四部委联合发文的形式下达, 足见国家对人工智能技术及应用的重视程度。这说明人工智能的意义已经超出本身, 成为创新创业国家战略成败的一部分。

参考文献

- [1] F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957.
- [2] P. H. Chen, C. J. Lin, and B. Schölkopf, A tutorial on v-support vector machines, Appl. Stoch. Models. Bus. Ind. 2005, 21, 111-136.
- [2] J Friedman, R Tibshirani, Special Invited Paper. Additive Logistic Regression: A Statistical View of Boosting: Discussion, Annals of Statistics, 2000.
- [4] K He, X Zhang, S Ren, J Sun, Deep Residual Learning for Image Recognition, Computer Science, 2015.
- [5] HINTON, Prof. Geoffrey Everest. Who's Who. (online Oxford University Press ed.). A & C Black, an imprint of Bloomsbury Publishing plc. 2015.
- [6] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In NIPS, 2012.
- [7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In CVPR, 2015.
- [8] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.

2

第2章 深度学习

深度学习 (Deep Learning) 是机器学习 (Machine Learning) 研究中的一个新领域, 是具有多隐含层的神经网络结构。深度学习通过组合低层特征形成更加抽象的高层表示属性或特征, 以发现数据的分布式特征表示。

本章将从生物意义上的神经网络模型, 引入人工神经网络模型, 以及 BP 神经网络和卷积神经网络, 最后介绍了深度学习的当前的主流框架。

2.1 神经网络模型

2.1.1 人脑视觉机理

1981 年的诺贝尔医学奖颁发给了 David Hubel、Torsten Wiesel 和 Roger Sperry。David Hubel 是一个出生在加拿大的美国神经生物学家, 他和 Torsten Wiesel 的主要贡献是“发现了视觉系统的信息处理”, 即可视皮层是分级的^[1], 如图 2-1 所示。

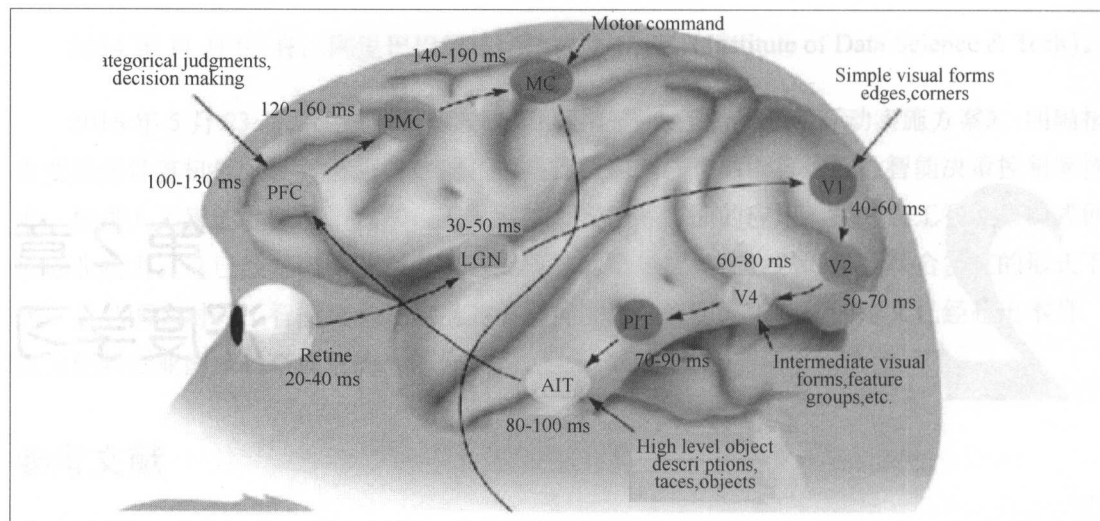


图 2-1 人脑可视皮层分级

1958 年, David Hubel 和 Torsten Wiesel 在 John Hopkins University, 研究瞳孔区域与大脑皮层神经元的对应关系。他们在猫的后脑头骨上, 开了一个 3 毫米的小洞, 向洞里插入电极, 测量神经元的活跃程度。

然后, 他们在小猫的眼前, 展现各种形状、各种亮度的物体。并且, 在展现每一件物体时, 还改变物体放置的位置和角度。他们期望通过这个办法, 让小猫瞳孔感受不同类型、不同强度的刺激。

之所以做这个试验, 目的是证明一个猜测。位于后脑皮层的不同视觉神经元, 与瞳孔所受刺激之间, 存在某种对应关系。一旦瞳孔受到某一种刺激, 后脑皮层的某一部分神经元就会活跃。经历了很多天反复的枯燥的试验, 同时牺牲了若干只可怜的小猫, David Hubel 和 Torsten Wiesel 发现了一种被称为“方向选择性细胞 (Orientation Selective Cell)”的神经元细胞。当瞳孔发现了眼前的物体的边缘, 而且这个边缘指向某个方向时, 这种神经元细胞就会活跃。

这个发现激发了人们对于神经系统的进一步思考。神经-中枢-大脑的工作过程, 或许是一个不断迭代、不断抽象的过程。这里的关键词有两个: 一个是抽象, 一个是迭代。从原始信号, 做低级抽象, 逐渐向高级抽象迭代。

人类的逻辑思维，经常使用高度抽象的概念。例如，从原始信号摄入开始（瞳孔摄入像素 Pixels），接着做初步处理（大脑皮层某些细胞发现边缘和方向），然后抽象（大脑判定眼前的物体的形状是圆形的），然后进一步抽象（大脑进一步判定该物体是个气球）。

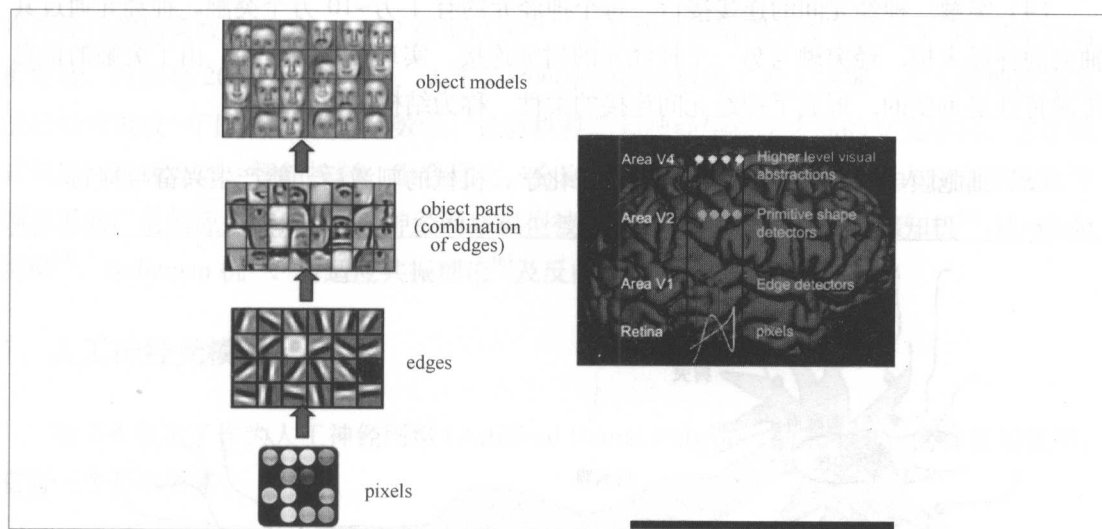


图 2-2 人脑视觉机制

这个生理学的发现，促成了计算机人工智能在 40 年后的突破性发展。总的来说，人的视觉系统的信息处理是分级的。从低级的 V1 区提取边缘特征，再到 V2 区的形状或者目标的部分内容，再到更高层，形成整个目标和目标的行为，如图 2-2 所示。也就是说高层的特征是低层特征的组合，从低层到高层的特征表示越来越抽象，越来越能表现语义或者意图。而抽象层面越高，存在的可能猜测就越少，就越利于分类。例如，单词集合和句子的对应是多对一的，句子和语义的对应又是多对一的，语义和意图的对应还是多对一的，这是一个层级体系。

2.1.2 生物神经元

生物神经元的结构如图 2-3 所示，包括以下部分。

(1) 细胞体：细胞核、细胞质和细胞膜。

(2) 树突：胞体短而多分枝的突起。相当于神经元的输入端。

(3) 轴突：胞体上最长枝的突起，也称神经纤维。端部有很多神经末梢传出神经冲动。

(4) 突触：神经元间的连接接口，每个神经元约有 1 万~10 万个突触。神经元通过其轴突的神经末梢，经突触与另一个神经元的树突连接，实现信息的传递。由于突触的信息传递特性是可变的，形成了神经元间连接的柔性，称为结构的可塑性。

(5) 细胞膜电位：神经细胞在受到电、化学、机械的刺激后，能产生兴奋与抑制。

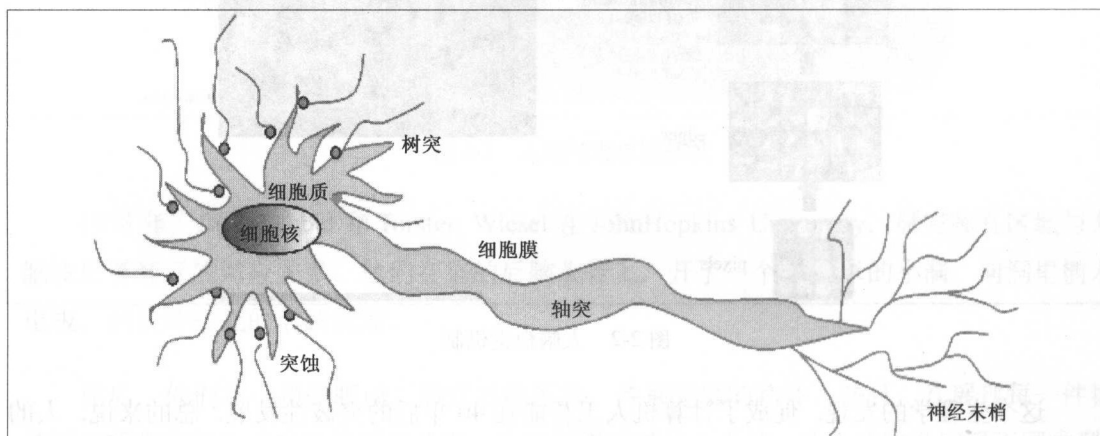


图 2-3 生物神经元结构

生物神经元功能如下。

(1) 兴奋与抑制：当传入神经元冲动，经整合使细胞膜电位升高，超过动作电位的阈值时，为兴奋状态，产生神经冲动，由轴突经神经末梢传出。当传入神经元的冲动，经整合，使细胞膜电位降低，低于阈值时，为抑制状态，不产生神经冲动。

(2) 学习与遗忘：由于神经元结构的可塑性，突触的传递作用可增强与减弱，因此，神经元具有学习与遗忘的功能。

2.1.3 人工神经网络

人工神经网络是在现代神经科学的基础上提出和发展起来的,旨在反映人脑结构及功能的一种抽象数学模型。自 1943 年美国心理学家 W. McCulloch 和数学家 W.A Pitts 提出形式神经元的抽象数学模型——MP 模型以来^[2],人工神经网络理论技术经过了 50 多年的曲折发展。特别是 20 世纪 80 年代,人工神经网络的研究取得了重大进展,有关的理论和方法已经发展成一门介于物理学、数学、计算机科学和神经生物学之间的交叉学科。它在模式识别、图像处理、智能控制、组合优化、金融预测与管理、通信、机器人及专家系统等领域得到广泛的应用,有 40 多种神经网络模型被提出,其中比较著名的有感知机^[3]、Hopfield 网络^[4]、Boltzman 机^[5]、自适应共振理论^[6]及反向传播网络 (BP)^[7]等。

1. 人工神经元模型

图 2-4 显示了作为人工神经网络 (Artificial Neural Network) 的基本单元的神经元模型,它有三个基本要素。

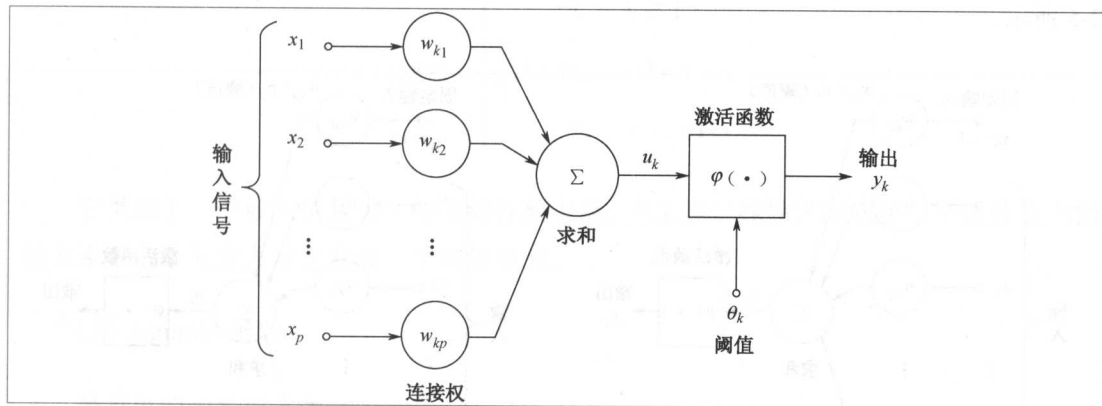


图 2-4 人工神经元结构

(1) 一组连接 (对应生物神经元的突触), 连接强度由各连接上的权值表示, 权值为正表示激活, 为负表示抑制。

(2) 一个求和单元, 用于求取各输入信号的加权和 (线性组合)。

(3) 一个非线性激活函数, 起非线性映射作用并将神经元输出幅度限制在一定范围内 (一般限值在 $[0,1]$ 或 $[-1,1]$ 之间)。

(4) 阈值 θ_k (或偏置 $b_k = -\theta_k$)。

以上作用可分别以数据公式表达出来:

$$u_k = \sum_{j=1}^p w_{kj} x_j, \quad v_k = u_k - \theta_k, \quad y_k = \varphi(v_k)$$

式中 x_1, x_2, \dots, x_p 为输入信号, $w_{k1}, w_{k2}, \dots, w_{kp}$ 为神经元 k 之权值, u_k 为线性组合结果, θ_k 为阈值, $\varphi(\cdot)$ 为激活函数, y_k 为神经元 k 的输出。

若把输入的维数增加一维, 则可把阈值 θ_k 包括进去。例如:

$$v_k = \sum_{j=0}^p w_{kj} x_j, \quad y_k = \varphi(v_k)$$

此处增加了一个新的连接, 其输入为 $x_0 = -1$ (或 $+1$), 权值为 $w_{k0} = \theta_k$ (或 b_k), 如图 2-5 所示。

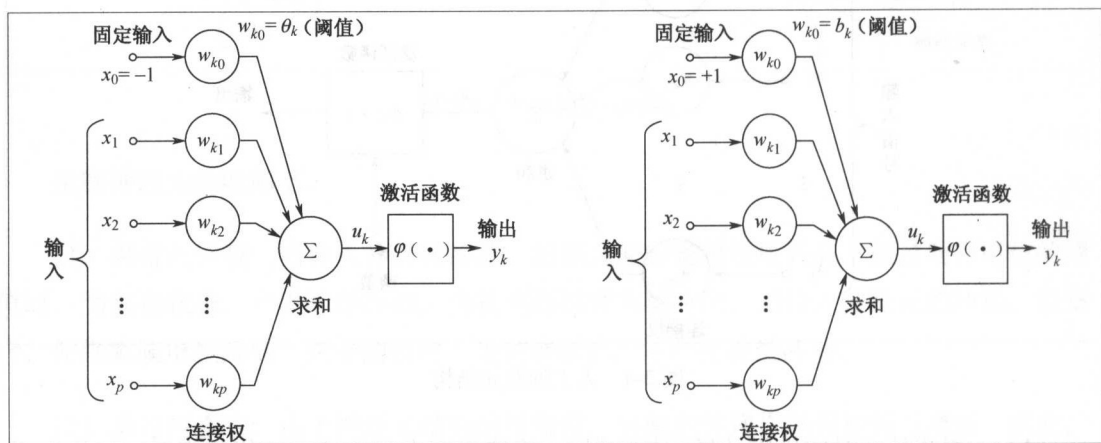


图 2-5 人工神经元数学表达

2. 激活函数

激活函数 $\varphi(\bullet)$ 有以下几种。

(1) 阈值函数

$$\varphi(v) = \begin{cases} 1, & v \geq 0 \\ 0, & v < 0 \end{cases}$$

即阶跃函数。这时相应的输出 y_k 为

$$y_k = \begin{cases} 1, & v_k \geq 0 \\ 0, & v_k < 0 \end{cases}$$

其中 $v_k = \sum_{j=1}^p w_{kj} x_j - \theta_k$ ，常称此种神经元为 M-P 模型。

(2) 分段线性函数

$$\varphi(v) = \begin{cases} 1, & v \geq 1 \\ \frac{1}{2}(1+v), & -1 < v < 1 \\ 0, & v \leq -1 \end{cases}$$

它类似于一个放大系数为 1 的非线性放大器，当工作于线性区时它是一个线性组合器，放大系数趋于无穷大时它变成一个阈值单元。

(3) sigmoid 函数

最常用的函数形式为

$$\varphi(v) = \frac{1}{1 + \exp(-\alpha^v)}$$

参数 $\alpha > 0$ 可控制其斜率。另一种常用的是双曲正切函数。

$$\varphi(v) = \tanh\left(\frac{v}{2}\right) = \frac{1 - \exp(-v)}{1 + \exp(-v)}$$

这类函数具有平滑和渐近性，并保持单调性。

2.2 BP 神经网络

通过以上章节我们知道人脑对信息的传递和对外界刺激所产生的反应都是由神经元控制的，人脑由上百亿个这样的神经元构成。这些神经元之间并不孤立而且联系很密切，每个神经元平均与几千个神经元相连接，因此构成了人脑的神经网络。刺激在神经网络中的传播是遵循一定规则的，一个神经元并非每次接到其他神经传递过来的刺激都产生反应。它首先会将与其相邻的神经元传来的刺激进行积累，到一定的时候产生自己的刺激并将其传递给一些与它相邻的神经元。这样工作的百亿个神经元构成了人脑对外界进行的反应。而人脑对外界刺激的学习机制就是通过调节这些神经元之间联系及其强度。当然，以上说的是对人脑真正神经工作的一种简化的生物模型，利用这种简化的生物模型可以将它推广至机器学习中来，并把它描述成人工神经网络。BP 神经网络就是其中的一种。

BP (Back Propagation) 网络是由 Rinehart 和 McClelland^[7]为首的科学家小组于 1986 年提出的，是一种按误差逆传播算法训练的多层前馈网络，是目前应用最广泛的神经网络模型之一。BP 网络能学习和存储大量的输入-输出模式映射关系，而无须事前揭示描述这种映射关系的数学方程。它的学习规则是使用梯度下降法，通过反向传播来不断调整网络的权值和阈值，使网络的误差平方和最小。BP 神经网络模型拓扑结构包括输入层 (Input)、隐层 (Hide layer) 和输出层 (Output layer)。

2.2.1 BP 神经元

图 2-6 给出了第 j 个基本 BP 神经元 (节点)，它只模仿了生物神经元所具有的三个最基本也是最重要的功能：加权、求和与转移。其中 $x_1, x_2, \dots, x_i, \dots, x_n$ 分别代表来自神经元 $1, 2, \dots, i, \dots, n$ 的输入； $w_{j1}, w_{j2}, \dots, w_{ji}, \dots, w_{jn}$ 则分别表示神经元 $1, 2, \dots, i, \dots, n$ 与第 j 个神经元的连接强度，即权值； b_j 为阈值， $f(\bullet)$ 为传递函数， y_j 为第 j 个神经元的输出。

第 j 个神经元的净输入值 S_j 为:

$$S_j = \sum_{i=1}^n w_{ji} x_i + b_j = W_j X + b_j$$

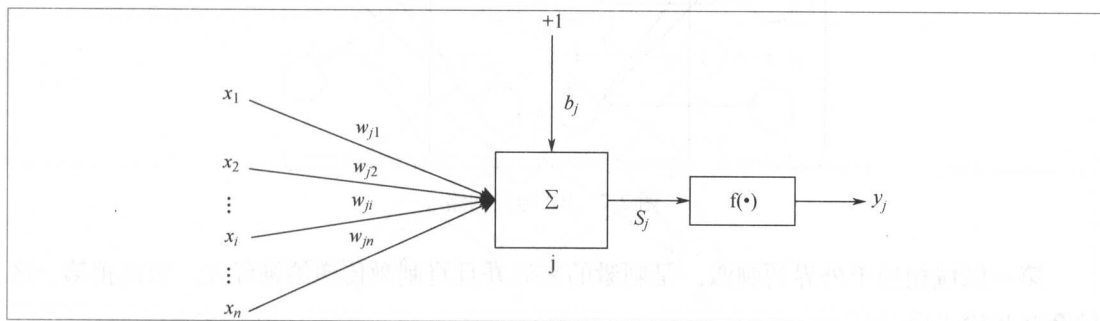


图 2-6 BP 神经元

其中: $X = [x_1, x_2, \dots, x_i, \dots, x_n]^T$, $W_j = [w_{j1}, w_{j2}, \dots, w_{ji}, \dots, w_{jn}]$, 若视 $x_0 = 1$, $w_{j0} = b_j$, 既令 X 及 W_j 包括 x_0 及 w_{j0} , 则 $X = [x_0, x_1, x_2, \dots, x_i, \dots, x_n]^T$, $W_j = [w_{j0}, w_{j1}, w_{j2}, \dots, w_{ji}, \dots, w_{jn}]$ 。于是节点 j 的净输入 S_j 可表示为:

$$S_j = \sum_{i=0}^n w_{ji} x_i = W_j X$$

净输入 S_j 通过传递函数 (Transfer Function) $f(\bullet)$ 后, 便得到第 j 个神经元的输出 y_j :

$$y_j = f(s_j) = f\left(\sum_{i=0}^n w_{ji} \cdot x_i\right) = F(W_j X)$$

其中 $f(\bullet)$ 是单调上升函数, 而且必须是有界函数, 因为细胞传递的信号不可能无限增加, 必有一最大值。

2.2.2 BP 神经网络构成

BP 神经网络用最直观的图形表示如图 2-7 所示。

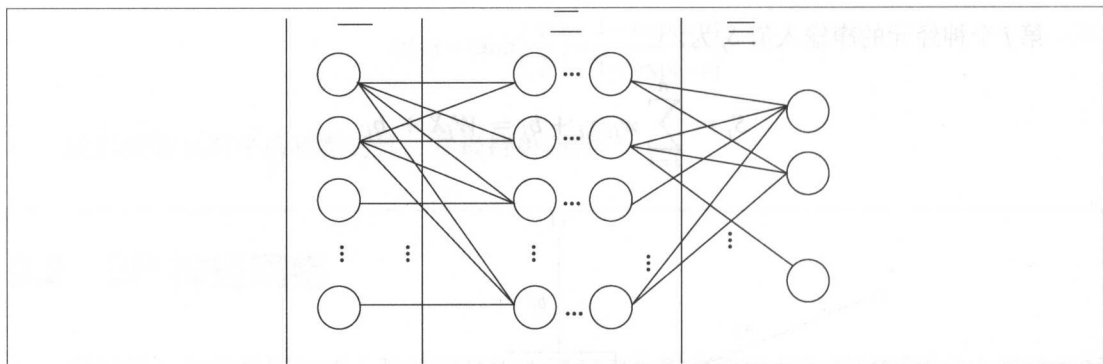


图 2-7 BP 神经网络

第一区域相当于外界的刺激，是刺激的来源并且将刺激传递给神经元，因此把第一区域命名为输入层。

第二区域，表示神经元相互之间传递刺激相当于人脑里面，因此把第二区命名为隐藏层。

第三区域，表示神经元经过多层次相互传递后对外界的反应，因此把第三区域命名为输出层。

简单的描述就是，输入层将刺激传递给隐藏层，隐藏层通过神经元之间联系的强度（权重）和传递规则（激活函数）将刺激传到输出层，输出层整理隐藏层处理后的刺激产生最终结果。若有正确的结果，那么将正确的结果和产生的结果进行比较，得到误差，再逆推对神经网络中的链接权重进行反馈修正，从而来完成学习的过程。这就是 BP 神经网的反馈机制，也正是 BP (Back Propagation) 名字的来源，即运用向后反馈的学习机制，来修正神经网络中的权重，最终达到输出正确结果的目的。

BP 算法由数据流的前向计算（正向传播）和误差信号的反向传播两个过程构成。正向传播时，传播方向为输入层→隐层→输出层，每层神经元的状态只影响下一层神经元。若在输出层得不到期望的输出，则转向误差信号的反向传播流程。通过这两个过程的交替进行，在权向量空间执行误差函数梯度下降策略，动态迭代搜索一组权向量，使网络误差函数达到最小值，从而完成信息提取和记忆过程。

2.2.3 正向传播

设 BP 网络的输入层有 n 个节点，隐层有 q 个节点，输出层有 m 个节点，输入层与隐层之间有权值为 v_{ki} ，隐层与输出层之间的权值为 w_{jk} ，如图 2-8 所示。

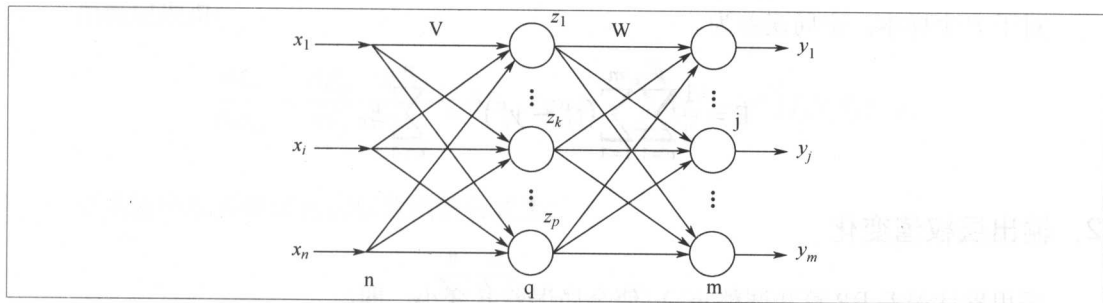


图 2-8 三层神经网络的拓扑结构

隐层的传递函数为 $f_1(\bullet)$ ，输出层的传递函数为 $f_2(\bullet)$ ，则隐层节点的输出为（将阈值写入求和项中， $k=1,2,\dots,p$ ）：

$$z_k = f_1 \left(\sum_{i=1}^n v_{ki} x_i \right)$$

输出层节点的输出为 ($j=1,2,\dots,m$)：

$$y_j = f_2 \left(\sum_{k=1}^q w_{jk} z_k \right)$$

至此 BP 网络就完成了 n 维空间向量对 m 维空间的近似映射。

2.2.4 反向传播

1. 定义误差函数

输入 P 个学习样本，用 $x^1, x^2, \dots, x^v, \dots, x^p$ 来表示。第 v 个样本输入到网络后得到输出 y_j^v ($j=1,2,\dots,m$)。采用平方型误差，于是得到第 v 个样本的误差 E_v ：

$$E_v = \frac{1}{2} \sum_{j=1}^m (t_j^v - y_j^v)^2$$

其中, t_j^v 为期望输出。

对于 P 个样本, 全局误差为:

$$E = \frac{1}{2} \sum_{v=1}^p \sum_{j=1}^m (t_j^v - y_j^v)^2 = \sum_{v=1}^p E_v$$

2. 输出层权值变化

采用累计误差 BP 算法调整 w_{jk} , 使全局误差 E 变小, 即

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}} = -\eta \frac{\partial}{\partial w_{jk}} \left(\sum_{v=1}^p E_v \right) = \sum_{v=1}^p \left(-\eta \frac{\partial E_v}{\partial w_{jk}} \right)$$

其中 η 为学习率 (learningrate)。

定义误差信号为:

$$\delta_{yj} = -\frac{\partial E_v}{\partial S_j} = -\frac{\partial E_v}{\partial y_j} \cdot \frac{\partial y_j}{\partial S_j}$$

其中第一项:

$$\frac{\partial E_v}{\partial y_j} = -\frac{\partial}{\partial y_j} \left[\frac{1}{2} \sum_{j=1}^m (t_j^v - y_j^v)^2 \right] = -\sum_{j=1}^m (t_j^v - y_j^v)$$

第二项:

$$\frac{\partial y_j}{\partial S_j} = f_2'(S_j)$$

是输出层传递函数的偏微分。

于是:

$$\delta_{yj} = \sum_{j=1}^m (t_j^v - y_j^v) f_2'(S_j)$$

由链定理得:

$$\frac{\partial E_v}{\partial w_{jk}} = \frac{\partial E_v}{\partial S_j} \cdot \frac{\partial S_j}{\partial w_{jk}} = -\delta_{yj} \cdot z_k = -\sum_{j=1}^m (t_j^v - y_j^v) f_2'(S_j) \cdot z_k$$

于是输出层各神经元的权值调整公式为:

$$\Delta w_{jk} = \sum_{v=1}^p \sum_{j=1}^m \eta (t_j^v - y_j^v) f_2'(S_j) \cdot z_k$$

3. 隐层权值变化

$$\Delta v_{ki} = -\eta \frac{\partial E}{\partial v_{ki}} = -\eta \frac{\partial}{\partial v_{ki}} \left(\sum_{v=1}^p E_v \right) = \sum_{v=1}^p \left(-\eta \frac{\partial E_v}{\partial v_{ki}} \right)$$

定义误差信号为:

$$\delta_{zk} = -\frac{\partial E_v}{\partial S_k} = -\frac{\partial E_v}{\partial Z_k} \cdot \frac{\partial Z_k}{\partial S_k}$$

其中第一项:

$$\frac{\partial E_v}{\partial Z_k} = -\frac{\partial}{\partial Z_k} \left[\frac{1}{2} \sum_{j=1}^m (t_j^v - y_j^v)^2 \right] = -\sum_{j=1}^m (t_j^v - y_j^v) \frac{\partial y_j}{\partial Z_k}$$

由链定理得:

$$\frac{\partial y_j}{\partial Z_k} = \frac{\partial y_j}{\partial S_j} \cdot \frac{\partial S_j}{\partial Z_k} = f_2'(S_j) w_{jk}$$

第二项:

$$\frac{\partial Z_k}{\partial S_k} = f_1'(S_k)$$

是隐层传递函数的偏微分。

于是:

$$\delta_{zk} = \sum_{j=1}^m (t_j^v - y_j^v) f_2'(S_j) w_{jk} f_1'(S_k)$$

由链定理得:

$$\frac{\partial E_v}{\partial v_{ki}} = \frac{\partial E_v}{\partial Z_k} \cdot \frac{\partial Z_k}{\partial v_{ki}} = -\delta_{zk} \chi_i = -\sum_{j=1}^m (t_j^v - y_j^v) f_2'(S_j) w_{jk} f_1'(S_k) \cdot \chi_i$$

从而得到隐层各神经元的权值调整公式为:

$$\Delta v_{ki} = \sum_{v=1}^p \sum_{j=1}^m \eta (t_j^v - y_j^v) f_2'(S_j) w_{jk} f_1'(S_k) \cdot \chi_i$$

2.3 卷积神经网络

卷积神经网络是人工神经网络的一种,已成为当前语音分析和图像识别领域的研究热点。它的权值共享网络结构使之更类似于生物神经网络,降低了网络模型的复杂度,减少了权值的数量。该优点在网络的输入是多维图像时表现得更为明显,使图像可以直接作为网络的输入,避免了传统识别算法中复杂的特征提取和数据重建过程。卷积网络是为识别二维形状而特殊设计的一个多层感知器,这种网络结构对平移、比例缩放、倾斜或者其他形式的变形具有高度不变性。

CNNs (Convolutional Neural Networks)^[10]是受早期的延时神经网络(TDNN)的影响。延时神经网络通过在时间维度上共享权值降低学习复杂度,适用于语音和时间序列信号的处理。

CNNs 是第一个真正成功训练多层网络结构的学习算法。它利用空间关系减少需要学习的参数数目以提高一般前向 BP 算法的训练性能。CNNs 作为一个深度学习架构提出,是为了最小化数据的预处理要求。在 CNNs 中,图像的一小部分(局部感受区域)作为层级结构的最低层的输入,信息再依次传输到不同的层,每层通过一个数字滤波器获得观测数据的最显著的特征。这个方法能够获取对平移、缩放和旋转不变的观测数据的显著特征,因为图像的局部感受区域允许神经元或者处理单元可以访问到最基础的特征,例如定向边缘或者角点。

2.3.1 卷积神经网络的历史

1962 年 Hubel 和 Wiesel 通过对猫视觉皮层细胞的研究^[1],提出了感受野(receptive field)的概念,1984 年日本学者 Fukushima 基于感受野的概念提出的神经认知机(neocognitron)^[8],可以看作是卷积神经网络的第一个实现网络,也是感受野这一概念在人工神经网络领域的首次应用。神经认知机将一个视觉模式分解成许多子模式(特征),然后进入分层递阶式相连的特征平面进行处理,它试图将视觉系统模型化,即使物体有位移或轻微变形的时候,也能完成识别。

通常神经认知机包含两类神经元,即承担特征抽取的 S-元和抗变形的 C-元。S-元中涉及两个重要参数,即感受野与阈值参数,前者确定输入连接的数目,后者则控制对特征子模式的反应程度。许多学者一直致力于提高神经认知机的性能的研究,在传统的神经认知机中,每个 S-元的感光区中由 C-元带来的视觉模糊量呈正态分布。如果感光区的边缘所产生的模糊效果要比中央来得大,则 S-元将会接受这种非正态模糊所导致的更大的变形容忍性。我们希望得到的是,训练模式与变形刺激模式在感受野的边缘与其中心所产生的效果之间的差异变得越来越大。为了有效地形成这种非正态模糊,Fukushima 提出了带双 C-元层的改进型神经认知机。

Van Ooyen 和 Niehuis 为提高神经认知机的区别能力引入了一个新的参数^[9]。事实上,该参数作为一种抑制信号,抑制了神经元对重复激励特征的激励。多数神经网络在权值中记忆训练信息。根据 Hebb 学习规则,某种特征训练的次数越多,在以后的识别过程中就越容易被检测。也有学者将进化计算理论与神经认知机结合,通过减弱对重复性激励特征

的训练学习，而使得网络注意那些不同的特征以助于提高区分能力。上述都是神经认知机的发展过程，而卷积神经网络可看作神经认知机的推广形式，神经认知机是卷积神经网络的一种特例。

2.3.2 卷积神经网络的网络结构

卷积神经网络是一个多层的神经网络，每层由多个二维平面组成，而每个平面由多个独立神经元组成，如图 2-9 所示。

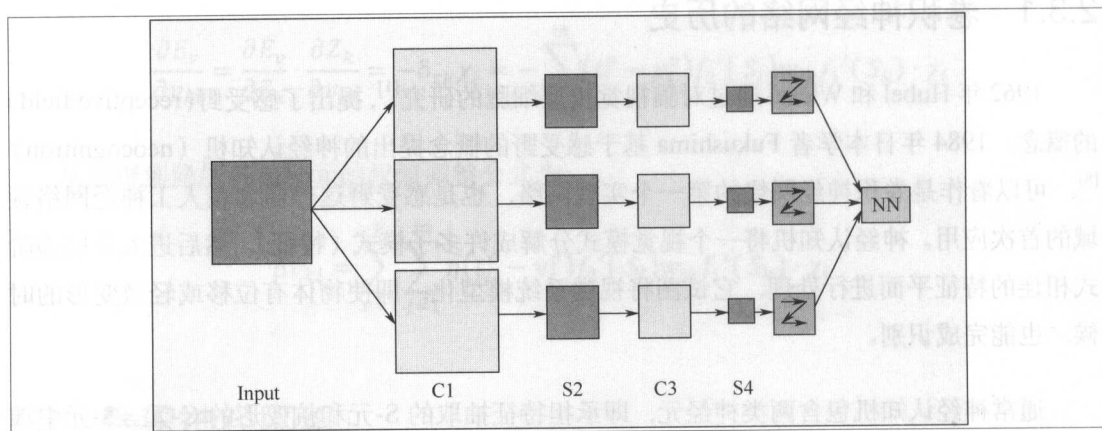


图 2-9 卷积神经网络的网络结构

输入图像通过三个可训练的滤波器和可加偏置进行卷积，卷积后在 C1 层产生三个特征映射图，然后特征映射图中每组的四个像素再进行求和，加权值，加偏置，通过一个 Sigmoid 函数得到三个 S2 层的特征映射图。这些映射图再经过滤波得到 C3 层。这个层级结构再和 S2 一样产生 S4。最终，这些像素值被光栅化，并连接成一个向量输入到传统的神经网络，得到输出。

一般地，C 层为特征提取层，每个神经元的输入与前一层的局部感受野相连，并提取该局部的特征，一旦该局部特征被提取后，它与其他特征间的位置关系也随之确定下来。S 层是特征映射层，网络的每个计算层由多个特征映射组成，每个特征映射为一个平面，平面上所有神经元的权值相等。特征映射结构采用影响函数核小的 sigmoid 函数作为卷积网络的激活函数，使得特征映射具有位移不变性。

此外，由于一个映射面上的神经元共享权值，因而减少了网络自由参数的个数，降低了网络参数选择的复杂度。卷积神经网络中的每一个特征提取层（C-层）都紧跟着一个用来求局部平均与二次提取的计算层（S-层），这种特有的两次特征提取结构使网络在识别时对输入样本有较高的畸变容忍能力。

2.3.3 局部感知

卷积神经网络有两种技术可以降低参数数目，第一种技术叫做局部感知野。一般认为人对外界的认知是从局部到全局的，而图像的空间联系也是局部的像素联系较为紧密，而距离较远的像素相关性则较弱。因此，每个神经元其实没有必要对全局图像进行感知，只需要对局部进行感知，然后在更高层将局部的信息综合起来就得到了全局的信息。网络部分连通的思想，也是启发于生物学里面的视觉系统结构。视觉皮层的神经元就是局部接受信息的（即这些神经元只响应某些特定区域的刺激）。如图 2-10 所示，左图为全连接，右图为局部连接。

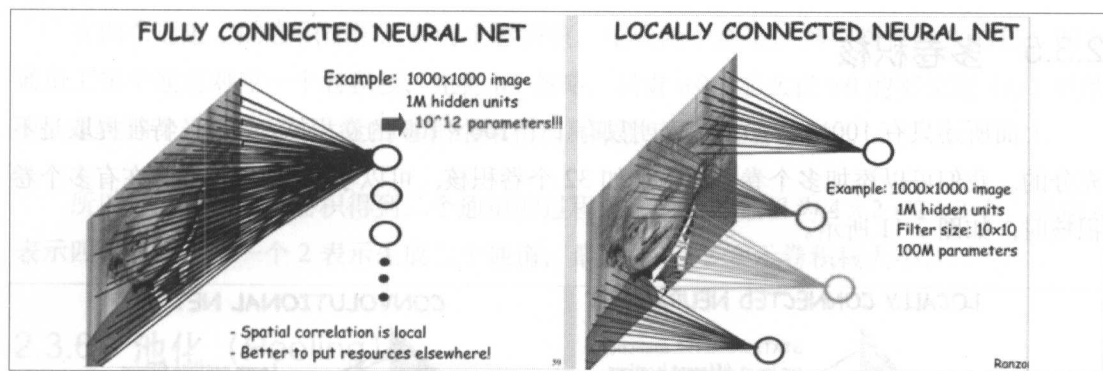


图 2-10 全连接与局部连接

在 2-10 右图中，假如每个神经元只和 10×10 个像素值相连，那么权值数据为 $1,000,000 \times 100$ 个参数，减少为原来的千分之一。而那 10×10 个像素值对应的 10×10 个参数，其实就相当于卷积操作。

2.3.4 参数共享

如果采用上述方法，参数仍然过多，那么就需要新的策略，即权值共享。在上面的局部连接中，每个神经元都对应 100 个参数，一共 1,000,000 个神经元，如果这 1,000,000 个神经元的 100 个参数都是相等的，那么参数数目就变为 100 了。

可以为这 100 个参数（也就是卷积操作）看成是提取特征的方式，该方式与位置无关。这其中隐含的原理则是：图像的一部分的统计特性与其他部分是一样的。这也意味着我们在这部分学习的特征也能用在另一部分上，所以对于这个图像上的所有位置，我们都能使用同样的学习特征。

更直观一些，当从一个大尺寸图像中随机选取一小块，比如说 8×8 作为样本，并且从这个小块样本中学习得到了一些特征，这时我们可以把从这个 8×8 样本中学习到的特征作为探测器，应用到这个图像的任意地方中去。特别地，我们可以用从 8×8 样本中所学习到的特征跟原本的大尺寸图像作卷积，从而对这个大尺寸图像上的任一位置获得一个不同特征的激活值。

2.3.5 多卷积核

上面所述只有 100 个参数时，表明只有 1 个 100×100 的卷积核，显然，特征提取是不充分的，我们可以添加多个卷积核，比如 32 个卷积核，可以学习 32 种特征。在有多个卷积核时，如图 2-11 所示。

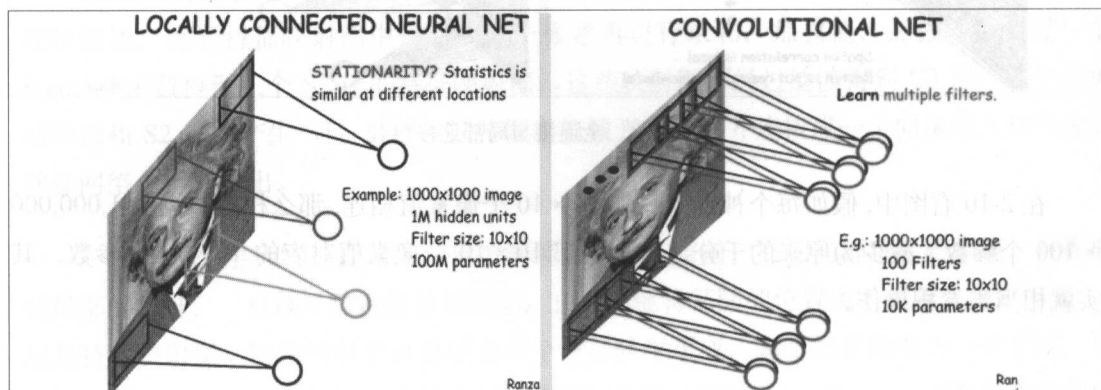


图 2-11 多卷积核

图 2-11 右图中对图片不同部分进行卷积，它们表明了不同的卷积核。每个卷积核都会将图像生成成为另一幅图像。比如两个卷积核就可以生成两幅图像，这两幅图像可以看作一张图像的不同通道。如图 2-12 所示。

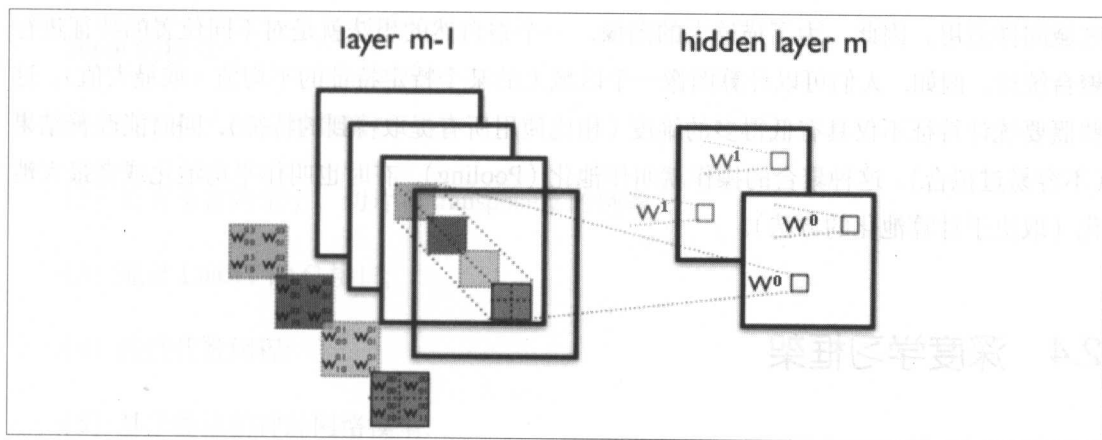


图 2-12 多卷积核

在四个通道上的卷积操作，有两个卷积核，生成两个通道。其中需要注意的是，四个通道上每个通道对应一个卷积核，先将 w_1 忽略，只看 w_0 ，那么在 w_0 的某位置 (i, j) 处的值，是由四个通道上 (i, j) 处的卷积结果相加再取激活函数值得到的。

所以，由四个通道卷积得到二个通道的过程中，参数的数目为 $4 \times 2 \times 2 \times 2$ 个，其中 4 表示四个通道，第一个 2 表示生成二个通道，最后的 2×2 表示卷积核大小。

2.3.6 池化 (Pooling)

在通过卷积获得了特征 (features) 之后，下一步我们希望利用这些特征去做分类。理论上讲，人们可以用所有提取得到的特征去训练分类器，例如 softmax 分类器，但这样做面临计算量的挑战。例如：对于一个 96×96 像素的图像，假设我们已经学习得到了 400 个定义在 8×8 输入上的特征，每一个特征和图像卷积都会得到一个 $(96 - 8 + 1) \times (96 - 8 + 1) = 7921$ 维的卷积特征。由于有 400 个特征，所以每个样例 (example) 都会得到一个 $7921 \times 400 = 3,168,400$ 维的卷积特征向量。学习一个拥有超过三百万特征输入的分类器十分不便，并且

容易出现过拟合 (over-fitting)。

为了解决这个问题,先回忆一下,我们之所以决定使用卷积后的特征,是因为图像具有一种“静态性”的属性,这也就意味着在一个图像区域中有用的特征极有可能在另一个区域同样适用。因此,为了描述大的图像,一个很自然的想法就是对不同位置的特征进行聚合统计。例如,人们可以计算图像一个区域上的某个特定特征的平均值(或最大值)。这些概要统计特征不仅具有低得多的维度(相比使用所有提取得到的特征),同时能改善结果(不容易过拟合)。这种聚合的操作就叫作池化(Pooling),有时也叫作平均池化或者最大池化(取决于计算池化的方法)。

2.4 深度学习框架

2.4.1 Caffe

Caffe 是一个清晰而高效的深度学习框架,作者是毕业于 UC Berkeley,目前在 Google 工作的贾扬清博士。

Caffe 的全称应该是 Convolutional Architecture for Fast Feature Embedding,它是开源的,核心语言是 C++,支持命令行、Python 和 MATLAB 接口,既可以在 CPU 上运行也可以在 GPU 上运行。License 是 BSD 2-Clause。

Caffe 可以应用在视觉、语音识别、机器人、神经科学和天文学领域。

Caffe 提供了一个完整的工具包,用来训练、测试、微调和部署模型。

Caffe 具有模块化、表示和实现分离、测试覆盖全面、接口丰富和预训练参考模型等特点。

Caffe 以四维数组 blobs 方式存储和传递数据,利用层之间计算结果传递每层的输入和输出。Caffe 模型是终端到终端的机器学习系统,其网络是一个有向无环层图。Caffe 通过快速、标准的随机梯度下降算法训练一个模型(Model)。

2.4.2 Torch

Torch 诞生已经有十多年，是一个广泛支持机器学习算法的科学计算框架，易于使用且高效，具有简单和快速的脚本语言 LuaJIT 和底层 C/CUDA 实现。

Torch 的特点：

- (1) 具有强大的 n 维数组；
- (2) 具有丰富的索引、切片和 transposing 的例程
- (3) 通过 LuaJIT 的 C 接口；
- (4) 线性代数例程；
- (5) 基于能量的神经网络模型；
- (6) 数值优化例程；
- (7) 支持快速高效的 GPU；
- (8) 可移植嵌入到 iOS、Android 和 FPGA 平台。

Torch 目标是通过极其简单的过程、最大的灵活性和速度建立自己的科学算法。Torch 有一个在机器学习领域的大型生态社区驱动库包，包括计算机视觉软件包、信号处理、并行处理、图像、视频、音频和网络等。

Torch 的核心是流行的神经网络，它使用简单的优化库，同时具有最大的灵活性，实现复杂的神经网络的拓扑结构。通过 CPU 和 GPU 等有效方式，可以建立神经网络和并行任意图。

Torch 广泛使用在许多学校的实验室，以及 Google (DeepMind)、Twitter、NVIDIA、AMD、英特尔和许多其他公司。

Facebook 开源了他们基于 Torch 的深度学习库包，这个版本包括 GPU 优化的大卷积网

(ConvNets) 模块, 以及稀疏网络, 这些通常被用在自然语言处理的应用中。ConvNet 模块包括 FFT-based 卷积层, 使用的是建立在 NVIDIA 的 CUFFT 库上自定义优化的 CUDA 内核。

2.4.3 Keras

Keras 是一个简约的、高度模块化的神经网络库, 是基于 Theano 的一个深度学习框架, 它的设计参考了 Torch, 用 Python 语言编写, 支持 GPU 和 CPU。其特点为:

- (1) 使用简单, 能够快速实现原理;
- (2) 支持卷积网络和递归网络, 以及两者的组合;
- (3) 无缝运行在 CPU 和 GPU 上;
- (4) 支持任意连接方式, 包括多输入多输出训练。

Keras 库与其他采用 Theano 库的区别是 Keras 的编码风格非常简约、清晰。它把所有的要点使用小类封装起来, 能够很容易地组合在一起并创造出一种全新的模型。

2.4.4 MXNet

MXNet 是一个轻量化分布式可移植的深度学习计算平台, 它支持多机多节点、多 GPU 的计算, 其 openMP+MPI/SSH+Cuda/Cudnn 的框架的计算速度很快, 且能够与分布式文件系统结合实现大数据的深度学习。MXNet 支持从单机到多 GPU、多集群的计算能力。其特点如下:

- (1) 基于赋值表达式建立计算图, 类似于 Tensorflow、Teano、Torch 和 Caffe;
- (2) 支持内存管理, 并对两个不交叉的变量重复使用同一内存空间;
- (3) 使用 C++ 实现, 并提供 C 风格的头文件。支持 Python、R、Julia、Go 和 Javascript;
- (4) 支持 Torch;

- (5) 支持移动设备端发布。

2.4.5 TensorFlow

TensorFlow 是 Google 基于 DistBelief 进行研发的第二代人工智能学习系统，其命名来源于本身的运行原理。Tensor（张量）意味着 N 维数组，Flow（流）意味着基于数据流图的计算，TensorFlow 为张量从图像的一端流动到另一端的计算过程。TensorFlow 是将复杂的数据结构传输至人工智能神经网络中进行分析 and 处理过程的系统。

TensorFlow 表达了高层次的机器学习计算，大幅简化了第一代系统，并且具备更好的灵活性和可延展性，可被用于语音识别或图像识别等多项机器深度学习领域。TensorFlow 对 2011 年开发的深度学习基础架构 DistBelief 进行了各方面的改进，它可在小到一部智能手机、大到数千台数据中心服务器的各种设备上运行。TensorFlow 完全开源，任何人都可以用。TensorFlow 的特点如下：

- (1) 高度的灵活性。TensorFlow 不是一个严格的“神经网络”库，只要能够将计算表示为一个数据流图，就可以使用 TensorFlow。

- (2) 较强可移植性。TensorFlow 可以在 CPU 和 GPU 上运行，也可以将模型在云端服务器或 Docker 容器里运行。

- (3) 自动求微分。TensorFlow 具有自动求微分的能力，只需要定义预测模型的结构，将结构和目标函数结合，并添加数据，TensorFlow 就能自动计算相关的微分导数。

- (4) 多语言支持。TensorFlow 支持 C++、Python、Go、Java、Lua 和 JavaScript 等语言。

2.4.6 CNTK

CNTK (Computational Network Toolkit) 是微软用于搭建深度神经网络的计算网络工具包，此项目已在 Github 上开源。CNTK 有一套极度优化的运行系统来训练和测试神经网络，它是以抽象的计算图形式构建的。CNTK 支持 CPU 和 GPU 模型。

CNTK 支持两种方式来定义网络：一种是使用“Simple Network Builder”，通过设置少量参数就能生成一个的标准神经网络；另一种是使用网络定义语言（NDL）。

CNTK 相比 Caffe、Theano、TensorFlow 等主流工具性能更强，灵活性也要好，可扩展性高。CNTK 支持 CNN、LSTM、RNN 等流行的网络结构，支持 CPU 和 GPU 模式。

CNTK 得到微软的支持，但目前 Bug 比较多，不太适合初学者学习。

2.4.7 Theano

Theano 是 BSD 许可证下发布的一个开源项目，是由 LISA（现 MILA）在加拿大魁北克的蒙特利尔大学开发的基于 Python 的深度学习框架，专门用于定义、优化、求值数学表达式，其效率比较高，适用于多维数组。

Python 的核心 Theano 是一个数学表达式的编译器。Theano 获取用户数据结构，使之成为一个使用 Numpy、高效本地库的非常高效的代码，并能在 CPU 或 GPU 上尽可能快地运行，其特点如下。

- (1) 紧密集成 Numpy。在 Theano 的编译函数中使用 `numpy.ndarray`。
- (2) 透明使用 GPU，使得执行数据密集型的计算速度高达 CPU 的 140 倍（仅对浮点数操作）。
- (3) 高效的符号分解。Theano 计算一个或多个输入函数的推导。
- (4) 速度和稳定性优化。即使 x 非常小也能正确得到 $\log(1+x)$ 的结果。
- (5) 动态生成 C 语言代码。计算表达式更快速。
- (6) 广泛的单元测试和自我验证。能检测和诊断许多类型的错误。

由于 Python 是为深度学习中处理大型神经网络算法所需的计算而专门设计的，其效率相比其他深度学习框架较弱，比较适合研究人员使用，不适合在线部署。

参考文献

- [1] Hubel, D. H., Wiesel, T. N., Receptive fields of single neurones in the cat's striate cortex, 1959.
- [2] W. McCulloch, W.A Pitts Logical Calculus of the Ideas Immanent in Nervous Activity, 1943.
- [3] ROSENBLATT F. The perceptron: a probabilistic model for information storage and organization in the brain[J]. Psychological Review, 1958, 65(6): 386.
- [4] Hopfield, J. J. and Tank, D. W. "Neural" computation of decisions in optimization problems. (1985) Biological Cybernetics 55, 141-146.
- [5] Ackley, A., Hinton, E., Sejnowski, T.: Boltzmann machines: Constraint satisfaction networks that learn. Technical Report CMU-CS- 84-119, Carnegie-Mellon University, Pittsburgh, 1984.
- [6] GA Carpenter, ORIGINAL CONTRIBUTION Fuzzy ART: Fast Stable Learning and Categorization of Analog Patterns by an Adaptive Resonance System.
- [7] BENGIOY. Learning deep architectures for AI [J]. Foundations and Trends in Machine Learning, 2009, 2 (1): 1-127.
- [8] FUKUSHIMA K, MIYAKE S. Neocognitron: a new algorithm for pattern recognition tolerant of deformations and shifts in position[J]. Pattern Recognition, 1982, 15 (6): 455-469.
- [9] A. van Ooyen, B. Nienhuis, Improving the convergence of the back-propagation algorithm, Neural Networks 5 (1992) 465-471.
- [10] VINCENTP, LAROCHELLEH, BENGIOY, et al. Extracting and Composing robust features with denoising autoencoders [C] //Proc of the 25th International Conference on Machine Learning. NewYork: ACMPress, 2008: 1096-1103.

3

第3章 Caffe 简介及其安装配置

Caffe^[1]是一个清晰而高效的深度学习框架，是纯粹的 C++/CUDA 架构，支持命令行、Python 和 MATLAB 接口；可以在 CPU 和 GPU 之间无缝切换，其作者是硕士毕业于清华大学、博士毕业于 UC Berkeley 大学的贾扬清。Caffe 问世至今，由于其使用简洁方便，执行效率高，在深度学习领域广受欢迎。

本章从介绍 Caffe 的基础开始，进一步介绍 Caffe 的使用环境，并详细介绍 Caffe 的安装与配置方法。

3.1 Caffe 是什么

Caffe 的全称是 Convolutional Architecture for Fast Feature Embedding，它是一个清晰、高效的深度学习框架，它是开源的，核心语言是 C++，它支持命令行、Python 和 MATLAB 接口，既可以在 CPU 上运行也可以在 GPU 上运行。

Caffe 是其作者贾扬清在 2013 年下半年利用空闲时间实现的。起初纯粹是一个业余的项目，但在 2013 年 12 月开源了之后，由于其高效、简洁，最终成为了一个在深度学习领域有影响力的学习框架。

从 2014 年开始, Caffe 开始吸引很多其他的用户和开发人员, 伯克利大学也建立了一个核心的 Caffe 团队, 特别是 NVIDIA 开始帮助 Caffe 团队做更多的加速, 伯克利大学也成立了 Berkeley Learning and Vision Center 来组织和吸引工业界的研究人员共同开发多个开源项目 (包括 Caffe)。Caffe 开始吸引世界各地的人来尝试和使用深度学习的技术。

在 Caffe 之前, 深度学习领域缺少一个完全公开所有的代码、算法和各种细节的框架, 导致很多研究人员和博士生需要一次又一次地重复实现相同的算法。而 Caffe 很好地解决了这个问题。

Caffe 的基本工作流程是设计建立在神经网络的一个简单假设, 所有的计算都是层的形式表示的, 网络中层做的事情就是输入数据, 然后输出计算以后的结果。比如卷积就是输入一个图像, 然后和这一层的参数 (filter) 做卷积, 最终输出卷积的结果。每层需要进行两种函数计算: 一种是 forward, 从输入计算到输出; 另一种是 backward, 从上层给的 gradient 来计算相对于输入层的 gradient。这两个函数实现了以后, 我们就可以把很多层连接成一个网络, 这个网络输入数据 (图像、语音或其他原始数据), 然后计算需要的输出 (比如识别的标签)。在训练的时候, 可以根据已有的标签计算 loss 和 gradient, 然后用 gradient 来更新网络中的参数。

当前的 Caffe 的 Master 分支只支持单机上的多 GPU 训练。2016 年 3 月浪潮发布了集群版 Caffe 计算框架, 它采用高性能计算行业成熟的 MPI 技术对 Caffe 版本进行数据并行的优化, 该并行 Caffe 计算框架完全保留了原始 Caffe 架构的特性。浪潮开发的集群并行 Caffe 计算框架已经在某超级计算机上进行部署并测试, 测试结果显示, 在保证正确率相同的情况下, 浪潮 Caffe 在 8 节点并行计算效率上提升了 10.7 倍, 大大提升了计算速度, 加速了业务的快速进行。相信在不久的将来, Caffe 的 Master 分支会支持集群的功能。

浪潮已经将其开发的集群并行版 Caffe 软件代码开源发布在 GitHub, 这将有助于更多的用户了解和应用这款软件, 加速深度学习的应用发展。有兴趣的读者可以试一下。

3.1.1 Caffe 的特点

1. 模块化

Caffe 设计之初就做到了尽可能的模块化, 允许对数据格式、网络层和损失函数进行扩展。

2. 表示和实现分离

Caffe 的模型 (Model) 定义是用 Protocol Buffer 语言写进配置文件的, 以任意有向无环图的形式, Caffe 支持网络架构。Caffe 会根据网络的需要来正确占用内存。通过一个函数调用, 实现 CPU 和 GPU 之间的切换。

3. 测试覆盖

在 Caffe 中, 每一个单一的模块都对应一个测试。

4. Python和MATLAB接口

同时提供 Python 和 MATLAB 接口。

5. 预训练参考模型

针对视觉项目, Caffe 提供了一些参考模型, 这些模型仅应用在学术和非商业领域, 它们的 License 不是 BSD。

3.1.2 Caffe 的架构

1. 数据存储

Caffe 通过 “Blobs” 即以 4 维数组的方式存储和传递数据。Blobs 提供了一个统一的内存接口, 用于批量图像 (或其他数据) 的操作和参数更新。Models 是以 Google Protocol Buffers

的方式存储在磁盘上的。大型数据存储在 LevelDB 数据库中。

2. 层

一个 Caffe 层 (Layer) 是一个神经网络层的本质, 它采用一个或多个 Blobs 作为输入, 并产生一个或多个 Blobs 作为输出。网络作为一个整体的操作, 层有两个关键职责: 前向传播, 需要输入并产生输出; 反向传播, 取梯度作为输出, 通过参数和输入计算梯度。Caffe 提供了一套完整的层类型。

3. 网络和运行方式

Caffe 保留所有的有向无环层图, 确保正确地进行前向传播和反向传播。Caffe 模型是终端到终端的机器学习系统。一个典型的网络开始于数据层, 结束于 loss 层。通过一个单一的开关, 使其网络运行在 CPU 或 GPU 上。在 CPU 或 GPU 上, 层会产生相同的结果。

4. 训练网络

Caffe 训练一个模型 (Model) 靠快速、标准的随机梯度下降算法。在 Caffe 中, 微调 (Fine tuning) 是一个标准的方法, 它适应于存在的模型、新的架构或数据。对于新任务, Caffe 微调旧的模型权重并按照需要初始化新的权重。

3.2 Caffe 的安装环境

3.2.1 Caffe 的硬件环境

用户采用 Caffe 进行深度学习训练, 需要一台适合自己需求的计算机。刚刚上手的初学者和基于研究性质的科学家所需要的计算机性能和配置并不相同。本节会详细阐述计算机硬件^[2]选择的要点。

1. CPU的选择

Caffe 支持 CPU 训练和 GPU 训练，所以读者在选择 Caffe 的硬件环境时必须考虑是应用 CPU 训练还是 GPU 训练。

如果是采用 CPU 训练，CPU 支持的线程数越多越好，因为 Caffe 本身显性地使用两个线程，一个线程用来从数据库中读取数据，另外一个线程用来执行 Forward 和 Backward。但由于神经网络计算中有大量的矩阵计算，而提供矩阵计算的科学计算库会大量运用多线程来增加计算速度，所以当单独采用 CPU 训练时，CPU 支持的核数和线程数越多越好。针对初学者而言，通常只会运行一些小型的网络，比如基于 Mnist 数据库的 LeNet 网络，一个 CPU 训练就足够了。

如果是采用 GPU 训练，则大量运算由 GPU 完成，CPU 只运行 Caffe 的两个线程，而当前的 CPU 至少也是双核双线程的。因此，CPU 的选择没有特殊需求，普通的 Intel i3 CPU 就能满足要求。即使选用线程数更多的 CPU 也无法大幅度加速训练，因为训练的时效取决于 GPU。

2. 内存的选择

应用 CPU 还是 GPU 训练，对内存的选择构成影响。

如果是采用 CPU 训练，由于在科学计算中会出现大量的内存读写操作，所以内存的速度至关重要，选择支持双通道的内存以及高频率的内存是有利于训练的。

如果是采用 GPU 训练，则由于大量的科学计算由 GPU 完成，CPU 和 GPU 之间通过 PCIe 通道交互。由于 PCIe3.0 DMA 速度远小于内存的传输速度，所以内存的频率高低无法影响 CPU 和 GPU 之间的交互速度。这样的话，在采用 GPU 训练下，内存频率不是重要影响因素。

由于深度学习算法中涉及大量的参数，另外深度训练的每个 batch 的原始数据都放在内存中，导致深度学习算法对内存的容量要求很高。一般来说，CPU 的内存容量至少和 GPU 的内存容量相当。建议初学者选择 CPU 的内存容量至少是 GPU 的内存容量的两倍。

如果是单独 CPU 训练，内存容量的大小和 GPU 训练的容量大小是一致的。

如果只是稍稍尝试一下深度学习，运行 Mnist 数据库的 LeNet 训练，2GB 内存就足够了。但如果您希望深层次地学习深度学习算法，那么 8GB 内存只是起步，建议配置 16GB 甚至 32GB 内存。

3. GPU 的选择

如果采用 GPU 训练，那么一款合适的显卡（GPU）至关重要。首先必须注意的是，因为 Caffe 只支持 Cuda 库，而 Cuda 库是 NVIDIA 显卡专用的，所以如果选择 Caffe 作为深度学习框架，请一定选购 NVIDIA 显卡。

如果预算足够，可以考虑购买 M40 这样的专为科学计算开发的顶级计算卡。但一般情况下，我们会选择通用显卡用于深度学习的 GPU 计算。比如，选用最新的 Pascal 架构的 GTX1080 或 GTX1070，显存 8GB。如果训练所需显存超过 8GB，对于单卡来说，可以选择 GTX Titan X，显存 12GB。对于一般小型神经网络来说，所需显存不会超过 4GB，可以选择 GTX980Ti、GTX980、GTX970 或 GTX960。GTX980Ti 拥有 6GB 显存，其他的为 4G 显存（注意 GTX970 的设计决定了其无法用足其 4GB 内存，本书不推荐）。

从运算速率来看，GTX1080 比 GTX1070 快 25% 左右，GTX1070 稍快于 GTX Titan X，大约是 GTX 960 的三倍。

对于有意向采用多 GPU 在 Caffe 框架下训练的读者，必须注意，Caffe 的确支持不同类型的网卡同时训练，但如果一张高速卡和一张低速卡一起训练，则最终的速度相当于两张低速卡一起训练的速度，即以低速卡的速度为准则。

4. 硬盘的选择

Caffe 采用单独线程异步方式从硬盘中顺序读取数据，所以正常情况下，机械硬盘的速度在 GPU 速度不是很快的情况下勉强够用。但当前 GPU 的速度越来越快，这要求硬盘的速度必须跟上，才不至于成为瓶颈。另外一些模型比如 R-CNN 会有大量的向硬盘写数据的要求。所以建议读者根据实际情况考虑固态硬盘（SSD）。至于硬盘容量，和训练所需的数

数据集密切相关,如果用 ImageSet 2012 数据集,仅下载下来的数据集压缩文件就需要 100GB 以上的硬盘空间。

5. 主板的选择

如果采用 GPU 训练,需要选择有 PCIe \times 16 的卡槽的主板。如果有意向采用多 GPU 训练,则选择的主板上要有多个 PCIe \times 16 的卡槽。

友情提醒: CPU 所支持的 PCIe 通道数和主板支持的 PCIe 通道数会对多 GPU 训练产生影响。如果主板支持的 PCIe 版本为 2.0,则对 GPU 和 CPU 之间的数据交换产生一定的影响,所以尽可能选择支持 PCIe 3.0 的主板。

6. 电源的选择

由于一般 GPU 的功耗都比较大,在选购电源时一定要仔细计算功率,以免出现长时间运行系统不稳定的情况。所需功率的一个简单计算方法是, CPU 加上 GPU 所需功率,再加上其他组件额外所需功率,以及作为电力峰值缓冲的 100W ~ 300W。由于深度学习训练根据模型的大小,所需训练时间变化很大,比如 VGGNet,可能需要在计算机上持续训练数周的时间。因此,建议选用合适功率的白金电源,以确保电源输出的稳定性。

7. 机箱的选择

首先注意的是显卡的长度,一般机箱的尺寸会对显卡的长度和显卡的数量造成限制,所以请选购机箱时要注意机箱支持的显卡最大长度和显卡数量。另外如果采用多 GPU 训练,机箱的散热情况也需要综合考虑。在整个系统功耗很大的情况下,可以考虑水冷机箱。

8. 散热的选择

散热十分重要,这可能成为一个瓶颈,比糟糕的硬件选择更能降低系统表现,因为 GPU 在过热的情况下会自动降频。

当 GPU 运行一个算法时,它会一直加速至最大极限,同时加速的还有能量消耗。但当

到达它的温度壁垒，通常是 80 摄氏度，它就会降低运行速度来避免打破临界温度。在保护 GPU 避免过热、情况安全的同时，也让 GPU 达到最佳状态。

但是，典型的电扇速度设置是被预编程的，完全不是为运行深度学习项目而设计的，所以，在开始深度学习程序后的几秒内就会达到其温度壁垒，结果是 GPU 的运行效果下降（几个百分比）。如果有好几个 GPU，由于 GPU 之间会相互加热，那么整体效果的下降幅度就会非常明显（10%~25%）。所以建议安装多个机箱风扇形成风道进行散热。如果是多 GPU 的情况，在预算足够时，建议安装水冷。

3.2.2 Caffe 的软件环境

Caffe 当前支持 Linux、OS X 和 Windows 三大操作系统。但对于入门的 Caffe 使用者来说，建议在 Linux 操作系统下安装使用 Caffe。本书选用的是 Ubuntu 14.04 系统，此系统确保是和 Caffe 兼容的。请读者注意，如果安装 Ubuntu 系统，请选择安装 64 位系统，因为 Caffe 的代码在编写时默认 Integer 为 64 位。如果安装 32 位操作系统会导致在 32 位操作系统上安装的 Caffe 在某些情况下运行会出错。

Caffe 的安装独立性较差，有大量的依赖库，这也是 Caffe 在使用群体中最大的诟病之一。值得庆幸的是，虽然 Caffe 的安装由于依赖库的缘故，比较困难，但一旦安装完毕，Caffe 的使用非常简便。

Caffe 提供 MATLAB 接口。所以习惯使用 MATLAB 的读者可安装 MATLAB 软件与 Caffe 对接。本书建议用户安装 MATLAB，因为一些免费深度学习应用，比如 R-CNN 中的大量代码是基于 MATLAB 开发的。

Caffe 提供 Python 接口。本书建议安装 Anaconda 软件（Anaconda 软件是一个支持科学计算的 Python 集成环境）与 Caffe 对接。不想安装 Anaconda 软件的用户也可以直接使用 Ubuntu 环境内置的 Python 环境，但需要额外安装 Numpy 等 Python 库。如果安装 Anaconda，则默认采用 Anaconda 自带的 Python 环境，整个环境中已经包括了科学计算所需要的各种 Python 库，例如 Numpy、Pandas 等。本书选用的是 Anaconda2 64bit 的 Linux 版本，在 Anaconda 的网站上同时提供了 Anaconda2 和 Anaconda3，Anaconda2 包含的是 Python2.7，而 Anaconda3

包含的是 Python3.5。虽然 Caffe 只需要修改 Makefile.config 就能适应 Andconda2 和 Anaconda3，但建议读者安装 Anaconda2，此版本与当前的 Caffe 版本以及各依赖库兼容性最好。笔者在使用 Anaconda3 时出现了 Python3.5 和 libboost 版本不兼容的问题，虽然最后也得到了解决，但会耗费用户很多时间。

Caffe 需要 OpenCV 的支持，所以 OpenCV 的安装也是必需的。用户可根据自己的情况安装 OpenCV2 或 OpenCV3。

3.2.3 Caffe 的依赖库

1. Boost库

Boost 库是一个可移植、提供源代码的 C++库，作为标准库的后备，是 C++标准化进程的开发引擎之一。Boost 库由 C++标准委员会库工作组成员发起，其中有些内容有望成为下一代 C++标准库内容，它在 C++社区中影响甚大，是不折不扣的“准”标准库。Boost 由于其对跨平台的强调，对标准 C++的强调，与编写平台无关。Caffe 采用 C++作为主开发语言，其中大量的代码依赖于 Boost 库。

2. GFlags库

GFlags 是 Google 的一个开源的处理命令行参数的库，使用 C++开发，可以替代 getopt 函数。GFlags 与 getopt 函数不同，在 GFlags 中，标记的定义分散在源代码中，不需要列举在一个地方。Caffe 库采用 GFlags 库开发 Caffe 的命令行。

3. GLog库

GLog 是一个应用程序的日志库，提供基于 C++风格的流日志 API，以及各种辅助的宏。它的使用方式与 C++的 stream 操作类似。Caffe 运行时的日志输出依赖于 GLog 库。

4. LevelDB库

LevelDB 是 Google 实现的一个非常高效的 Key-Value 数据库。它是单进程的服务，性

能非常高。它是一个 C/C++ 编程语言的库。Caffe 支持两种数据库，其中之一为 LevelDB。

5. LMDB库

它是一个超级快、超级小的 Key-Value 数据存储服务，是由 OpenLDAP 项目的 Symas 开发的。使用内存映射文件，因此读取数据的性能跟内存数据库一样，其大小受限于虚拟地址空间的大小。Caffe 支持两种数据库，其中之一为 LMDB。

6. ProtoBuf库

Google Protocol Buffer（简称 ProtoBuf）是一种轻便高效的结构化数据存储格式，可以用于结构化数据的串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通信协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。

要使用 Protobuf 库，首先需要自己编写一个 .proto 文件，定义程序中需要处理的结构化数据，在 Protobuf 中，结构化数据被称为 Message。在一个 .proto 文件中可以定义多个消息类型。用 Protobuf 编译器（protoc.exe）将 .proto 文件编译成目标语言，会生成对应的 .h 文件和 .cc 文件，.proto 文件中的每一个消息有一个对应的类。

Caffe 使用起来非常简洁，很大程度上是由于 Caffe 采用 .proto 文件作为用户的输入接口。用户通过编写 .proto 文件定义网络模型和 Solver。

7. HDF5库

HDF（Hierarchical Data File）是美国国家高级计算应用中心（NCSA）为了满足各种研究领域研究需求而研制的一种能高效存储和分发科学数据的新型数据格式。它可以存储不同类型的图像和数码数据的文件格式，并且可以在不同类型的机器上传输，同时还有统一处理这种文件格式的函数库。Caffe 支持 HDF5 格式。

8. snappy库

它是一个 C++ 库，用来压缩和解压缩的开发包。它旨在提供高速压缩速度和合理的

压缩率。snappy 比 zlib 更快, 但文件相对要大 20%到 100%。Caffe 在数据处理时依赖于 snappy 库。

3.2.4 Caffe 开发环境的安装

由于本书篇幅有限, 这里主要介绍在 Ubuntu 环境下的 Caffe 开发环境的安装。在安装 Caffe 及其依赖库或软件前, 默认已安装了 Ubuntu 14.04 64bit 操作系统, 并拥有具备 CuDNN 能力的 NVIDIA GTX 显卡。本安装指南的系统配置如下:

- ubuntu 14.04 64bit OS
- Intel i7 CPU
- 32G 内存
- GTX1080 显卡

安装步骤如下。

1. 安装开发所需的依赖包

```
$sudo apt-get install build-essential  
$sudo apt-get install libprotobuf-dev libdevldevelldb-dev libsnappy-e-dev  
libopencv-dev libboost-all-dev  
$sudo apt-get install libhdf5-serial-dev libgflags-dev libgoogle-glog-dev  
liblmbd-dev protobuf-compiler
```

2. 安装CUDA7.5

NVIDIA 官方网站提供两种方法来安装 CUDA, 分别如下。

- run 安装: 读者可以从 NVIDIA 下载对应版本的 run 安装包安装。
- deb 安装: 读者可以从 NVIDIA 下载 deb 安装包安装。

不建议读者用 run 安装, 因为 run 安装对 Linux 的版本、Linux kernel 版本、GCC 版本

都有严格要求，稍有不符，可能就会导致安装失败。

下面介绍 deb 安装方式，请依次键入以下命令：

```
$sudo dpkg -i cuda-repo-<distro>_<version>_<architecture>.deb
$sudo apt-get update
$sudo apt-get install cuda
```

安装完后，重启电脑使其生效。

要验证 CUDA 是否安装成功，请读者在 `usr/local/cuda` 下搜索 `sample` 所在目录（deb 方式会自动安装 `cuda sample`），进入此目录，键入以下命令：

```
$make all
```

在 `make` 完成后，找到 `device_query` 这个执行文件并运行。如果 `cuda` 安装成功，`device_query` 会显示 GPU 的相关信息。

3. 安装cuDNN

从 NVIDIA 官网下载 `cudnn-7.5-linux-x64-v5.0-ga.tgz`，并解压缩后安装，命令如下：

```
$tar -zxvf cudnn-7.5-linux-x64-v5.0-ga.tgz
$cd cuda
$sudo cp lib/lib* /usr/local/cuda/lib64/
$sudo cp include/cudnn.h /usr/local/cuda/include/
```

接下更新 cuDNN 库文件的软连接，命令如下：

```
$cd /usr/local/cuda/lib64/
$sudo chmod +r libcudnn.so.5.0.5
$sudo ln -sf libcudnn.so.5.0.5 libcudnn.so.5
$sudo ln -sf libcudnn.so.5 libcudnn.so
$sudo ldconfig
```

4. 安装BLAS（基本线性代数子库）

Caffe 支持三种 AtLas，分别是 MKL，AtLas，OpenBlas。

MKL 是有限制公开的，有条件的用户可以到 <https://software.intel.com/en-us/intel-mkl/>

上自行下载。AtLas、OpenBlas 是完全免费的，可以直接通过 apt-get 安装。由于 OpenBlas 的效率接近 MKL，比 AtLas 的效率要高很多，且没有 MKL 的安装限制，故采用 OpenBlas 安装。安装命令如下：

```
$sudo apt-get install openblas-dev
```

注意：安装 OpenBlas 后，为使其生效，需要修改 Caffe 的 Makefile.config 文件中 BLAS 的选项为 BLAS := open。

5. 安装OpenCV

OpenCV 有两种安装方式。

(1) 脚本安装：OpenCV2.4.10

- 下载安装脚本
- 进入目录 Install-OpenCV/Ubuntu/2.4
- 执行脚本 opencv2_4_10.sh

(2) 源代码安装

首先从网上下载 OpenCV 源代码，这里为 opencv-3.0.0.tar.gz，用 tar -zxvf opencv-3.0.0.tar.gz ./解压，然后进入解压完的 opencv 目录。

```
$make .
$make all
$make install
```

如果安装的是 opencv3 而不是 opencv2 的话，则需要修改 Caffe 的 Makefile.config 文件。

```
$sudo gedit /etc/ld.so.conf.d/opencv.conf
```

将以下内容添加到 Caffe 的 Makefile.config 文件的最后：

```
/usr/local/lib
```


接下来配置库:

```
$sudo ldconfig
```

更改环境变量:

```
$sudo gedit /etc/bash.bashrcs
```

在 `bash.bashrcs` 文件最后添加如下路径:

```
PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/usr/local/lib/pkgconfig
export PKG_CONFIG_PATH
```

保存退出后, 需要重新开启终端使配置生效。

6. 安装Caffe的Python环境

Caffe 官方网站推荐使用 Anaconda, 在 Anaconda 官方网站下载安装包切换到文件所在目录, 执行如下命令:

```
$bash Anaconda-2.3.0-Linux-x86_64.sh
```

注意: 后面的文件名按下载的版本号自行调整, 整个安装过程请选择默认。

然后, 添加 Anaconda Library Path。在 `/etc/ld.so.conf` 最后加入以下路径 (注意: 下面的 `username` 请读者根据当前环境自行替换)。

```
/home/username/anaconda/lib
```

在 `~/.bashrc` 最后添加下面的路径:

```
export LD_LIBRARY_PATH="/home/username/anaconda/lib:$LD_LIBRARY_PATH"
```

7. 安装Caffe的Python依赖库

访问并打开 Caffe 的 Github 网站, 下载 Caffe 源码包, 进入 `caffe-master` 下的 `python` 目录, 执行如下命令:

```
$for req in $(cat requirements.txt); do pip install $req; done
```

8. 编译Caffe

进入 caffe-master 目录, 复制一份 Makefile.config.examples, 命令如下:

```
$cp Makefile.config.example Makefile.config
```

修改其中的一些路径, 如果上述步骤中的配置一致, 都选默认路径, 那么配置文件应该如下:

```
## Refer to http://caffe.berkeleyvision.org/installation.html
# Contributions simplifying and improving our build system are welcome!

# cuDNN acceleration switch (uncomment to build with cuDNN).
USE_CUDNN := 1

# CPU-only switch (uncomment to build without GPU support).
CPU_ONLY := 1

# To customize your choice of compiler, uncomment and set the following.
# N.B. the default for Linux is g++ and the default for OSX is clang++
CUSTOM_CXX := g++

# CUDA directory contains bin/ and lib/ directories that we need.
CUDA_DIR := /usr/local/cuda
# On Ubuntu 14.04, if cuda tools are installed via
# "sudo apt-get install nvidia-cuda-toolkit" then use this instead:
# CUDA_DIR := /usr

# CUDA architecture setting: going with all of them.
# For CUDA < 6.0, comment the *_50 lines for compatibility.
CUDA_ARCH := -gencode arch=compute_20,code=sm_20 \
              -gencode arch=compute_20,code=sm_21 \
              -gencode arch=compute_30,code=sm_30 \
              -gencode arch=compute_35,code=sm_35 \
              -gencode arch=compute_50,code=sm_50 \
              -gencode arch=compute_50,code=compute_50

# BLAS choice:
# atlas for ATLAS (default)
# mkl for MKL
# open for OpenBLAS
BLAS := atlas
# Custom (MKL/ATLAS/OpenBLAS) include and lib directories.
# Leave commented to accept the defaults for your choice of BLAS
# (which should work)!
# BLAS_INCLUDE := /path/to/your/blas
# BLAS_LIB := /path/to/your/blas
```

```

# Homebrew puts openblas in a directory that is not on the standard search
path
# BLAS_INCLUDE := $(shell brew --prefix openblas)/include
# BLAS_LIB := $(shell brew --prefix openblas)/lib

# This is required only if you will compile the matlab interface.
# MATLAB directory should contain the mex binary in /bin.
# MATLAB_DIR := /usr/local
# MATLAB_DIR := /Applications/MATLAB_R2012b.app

# NOTE: this is required only if you will compile the python interface.
# We need to be able to find Python.h and numpy/arrayobject.h.
#PYTHON_INCLUDE := /usr/include/python2.7 \
    /usr/lib/python2.7/dist-packages/numpy/core/include
# Anaconda Python distribution is quite popular. Include path:
# Verify anaconda location, sometimes it's in root.
ANACONDA_HOME := $(HOME)/anaconda
PYTHON_INCLUDE := $(ANACONDA_HOME)/include \
    $(ANACONDA_HOME)/include/python2.7 \
    $(ANACONDA_HOME)/lib/python2.7/site-packages/numpy/core/include \

# We need to be able to find libpythonX.X.so or .dylib.
#PYTHON_LIB := /usr/lib
PYTHON_LIB := $(ANACONDA_HOME)/lib

# Homebrew installs numpy in a non standard path (keg only)
# PYTHON_INCLUDE += $(dir $(shell python -c 'import numpy.core;
print(numpy.core.__file__)')/include
# PYTHON_LIB += $(shell brew --prefix numpy)/lib

# Uncomment to support layers written in Python (will link against Python
libs)
# WITH_PYTHON_LAYER := 1

# Whatever else you find you need goes here.
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib

# If Homebrew is installed at a non standard location (for example your home
directory) and you use it for general dependencies
# INCLUDE_DIRS += $(shell brew --prefix)/include
# LIBRARY_DIRS += $(shell brew --prefix)/lib

# Uncomment to use `pkg-config` to specify OpenCV library paths.
# (Usually not necessary -- OpenCV libraries are normally installed in one
of the above $LIBRARY_DIRS.)
# USE_PKG_CONFIG := 1

BUILD_DIR := build
DISTRIBUTE_DIR := distribute

```

```
# Uncomment for debugging. Does not work on OSX due to
https://github.com/BVLC/caffe/issues/171
# DEBUG := 1

# The ID of the GPU that 'make runtest' will use to run unit tests.
TEST_GPUID := 0

# enable pretty build (comment to see full commands)
Q ?= @
```

此时，执行编译命令：

```
$make all -j4
$make test
$make runtest
```

9. 编译Python wrapper

```
$make pycaffe
```

3.3 Caffe 接口

Caffe 深度学习框架支持多种编程接口，包括命令行、Python 和 Matlab，本节将介绍如果使用这些接口^[3]。

3.3.1 Caffe Python 接口

1. 接口功能

Caffe 提供 Python 接口，即 Pycaffe，具体实现在 caffe/python 文件夹内。在 Python 代码中 import caffe，可以 load models（导入模型）、forward and backward（前向、反向迭代）、handle IO（数据输入输出）、visualize networks（绘制 net）和 instrument model solving（自定义优化方法）。所有的模型数据、计算参数都是暴露在外、可供读写的。

(1) caffe.Net 是主要接口，负责导入数据、校验数据、计算模型。

(2) caffe.Classifier 用于图像分类。

- (3) caffe.Detector 用于图像检测。
- (4) caffe.SGDSolver 是露在外的 solver 的接口。
- (5) caffe.io 处理输入输出，数据预处理。
- (6) caffe.draw 可视化 net 的结构。
- (7) Caffe blobs 以 numpy ndarrays 的形式表示，方便而且高效。

2. Python安装

大多数的 Linux 系统默认都会安装 Python，可以通过命令查看是否已经安装 Python，以 Ubuntu 为例，输入命令如下：

```
$ python
Python 2.7.6 (default, Jun 11 2016, 09:30:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>print 'helloworld'
helloworld
>>>
```

以上输出信息表明安装成功，并提示当前系统中 Python 的版片号等信息。否则提示出错，则需要手动安装 Python，方法很简单，安装命令如下：

```
$ sudo apt-get install python
```

3. Pycaffe配置

- (1) 依赖库安装：

```
$ sudo apt-get install python-numpy python-scipy python-matplotlib
python-sklearn python-skimage python-h5py python-protobuf python-leveldb
python-networkx python-nose python-pandas python-gflags Cython ipython
$ sudo apt-get install protobuf-c-compiler protobuf-compiler
```

- (2) 依赖库编译：

```
$ cd ~/caffe
$ make pycaffe
```

(3) 添加 PYTHONPATH:

```
$ sudo gedit /etc/profile
#末尾添加: export PYTHONPATH=/path/to/caffe/python:$PYTHONPATH
#使用完整路径
$ source /etc/profile # 使之生效
```

其中“path/to/caffe/python”为 Python 的安装目录, 根据安装环境目录会不同, 请读者自行修改。

(4) 测试:

```
$ python
Python 2.7.6 (default, Jun 11 2016, 09:30:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import caffe
>>>
```

“import caffe”命令若没有提示出错, 则表明整个编译和安装成功。

4. Pycaffe使用示例

以下是一个使用 Caffe Python 接口的识别 LeNet 手写数字的示例:

```
1  import os
2  import sys
3  import numpy as np
4  import matplotlib.pyplot as plt
5  caffe_root = '/home/alex/caffe-master/'
6  sys.path.insert(0, caffe_root + 'python')
7  import caffe
8  MODEL_FILE = '/home/alex/caffe-master/examples/mnist/lenet.prototxt'
9  PRETRAINED =
'/home/alex/caffe-master/examples/mnist/lenet_iter_10000.caffemodel'
10 IMAGE_FILE = '/home/alex/caffe-master/examples/images/911.bmp'

11 input_image=caffe.io.load_image(IMAGE_FILE,color=False)
12 #print input_image
13 #net = caffe.Classifier(MODEL_FILE, PRETRAINED, channel_swap=(2,1,0),
image_dims=(28,28))
14 net = caffe.Classifier(MODEL_FILE, PRETRAINED)

15 #net = caffe.Classifier(MODEL_FILE, PRETRAINED, raw_scale=255,
image_dims=(28,28))
```

```
16 prediction=net.predict([input_image], oversample=False)
17 caffe.set_mode_cpu()
18 print 'predicted class:', prediction[0].argmax()
```

脚本第 7 行 import caffe 后, 即可调用 Caffe 的 Python 接口。

3.3.2 Caffe MATLAB 接口

MATLAB 接口 (Matcaffe) 在 caffe/matlab 目录的 caffe 软件包。在 matcaffe 的基础上, 可将 Caffe 整合到 MATLAB 代码中。

MATLAB 接口包括:

- (1) MATLAB 中创建多个网络结构。
- (2) 网络的前向传播 (Forward) 与反向传播 (Backward) 计算。
- (3) 网络中任意一层以及参数的存取。
- (4) 网络参数保存至文件或从文件加载。
- (5) blob 和 network 形状调整。
- (6) 网络参数编辑和调整。
- (7) 创建多个 solvers 进行训练。
- (8) 从 solver 快照 (Snapshots) 恢复并继续训练。
- (9) 访问训练网络 (Train nets) 和测试网络 (Test nets)。
- (10) 迭代后网络交由 MATLAB 控制。
- (11) MATLAB 代码融合梯度算法。

使用 Caffe 的 MATLAB 接口, 首先要进行编译, 命令如下:


```
$ cd ~/caffe
$ make all matcaffe
```

编译完成后,使用 `make mattest` 命令测试是否编译成功。为了方便使用,需要将 `matcaffe` 加入 Matlab 搜索路径,方法是在 Caffe 根目录启动 Matlab,在 Matlab 命令行运行如下命令:

```
Addpath ./matlab
```

在 `caffe/matlab/demo` 目录提供了 Caffe 的 MATLAB 的 demo 实例,打开 MATLAB,并切换当前目录到 `caffe-master/matlab/demo`,拷贝 `cat.jpg` 至当前目录,运行 `classification_demo.m` 即可得到图片的 1000 类输出。

3.3.3 Caffe 命令行接口

命令行接口 `Cmdcaffe` 是 Caffe 中用来训练模型、计算得分以及方法判断的工具。`Cmdcaffe` 存放在 `caffe/build/tools` 目录下。

1. caffe train

`caffe train` 命令用于模型学习,具体包括:

- (1) `caffe train` 带 `solver.prototxt` 参数完成配置。
- (2) `caffe train` 带 `snapshot mode_iter_1000.solverstate` 参数加载 solver snapshot。
- (3) `caffe train` 带 `weights` 参数 `model.caffemodel` 完成 Fine-tuning 模型初始化。

2. caffe test

`caffe test` 命令用于测试运行模型的得分,并且用百分比表示网络输出的最终结果,比如 `accuracy` 或 `loss` 作为其结果。测试过程中,显示每个 batch 的得分,最后输出全部 batch 的平均得分值。

3. caffe time

caffe time 命令用来检测系统性能和测量模型相对执行时间，此命令通过逐层计时与同步，执行模型检测。

参考文献

- [1] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093, 2014.
- [2] T. Dettmers, Which GPU(s) to Get for Deep Learning: My Experience and Advice for Using GPUs in Deep Learning, <http://timdettmers.com/2014/08/14/which-gpu-for-deep-learning/>, 2014.
- [3] Yangqing Jia, Evan Shelhamer, Caffe Tutorial, <http://caffe.berkeleyvision.org/tutorial/>, 2016.

4

第4章 Caffe 网络定义

上一章我们初步了解了 Caffe 的架构和安装环境, Caffe 作为一款经典的深度学习框架, 其在实现上有着清晰的分层网络定义, 并且具有较强的易读性、可移植性和结构化等特点。

本章先介绍基于 Caffe 的网络模型要素及构成, 然后介绍 Caffe 支持的数据库类型, 最后详细分析 Caffe 的 Net、Blob, 以及主要层的定义、功能和参数, 用大量的例子帮助读者理解 Caffe 的精髓之处。

4.1 Caffe 模型要素

Caffe 的模型需要两个重要的参数文件: 网络模型和参数配置, 分别是 *.prototxt 和 *.solver.prototxt 文件。

4.1.1 网络模型

Caffe 的网络模型文件定义了网络的每一层行为, 图 4-1 是利用 Caffe 中的 python/draw_net.py 画出的 LeNet 模型。

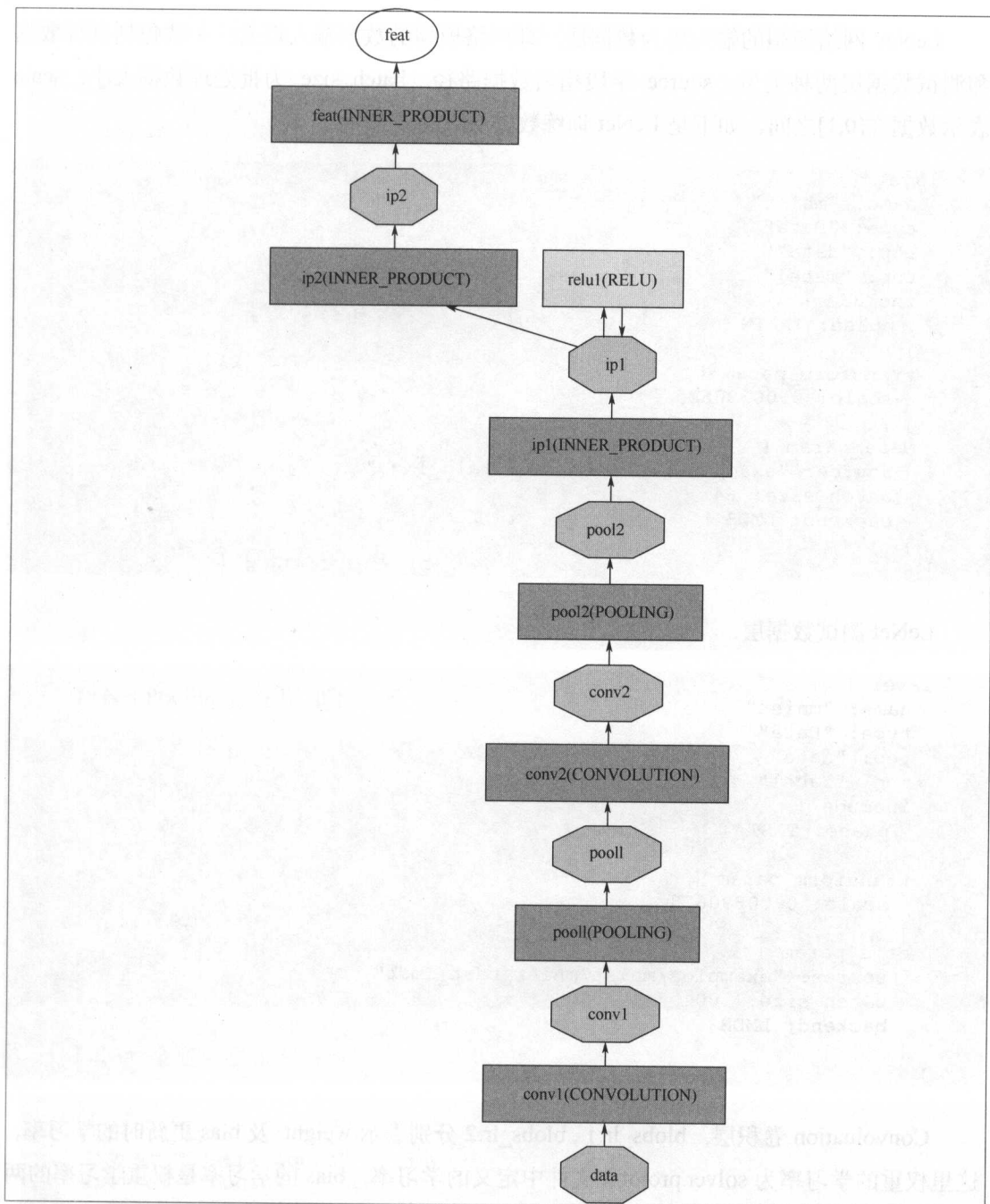


图 4-1 LeNet 网络模型

LeNet 网络模型的输入层为数据层，即网络模型的数据输入定义，一般包括训练数据和测试数据层两种类型。source 字段指名数据路径，batch_size 为批处理数据大小，scale 表示数据在[0,1]之间。如下是 LeNet 训练数据层：

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
```

LeNet 测试数据层：

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
```

Convolution 卷积层，blobs_lr:1, blobs_lr:2 分别表示 weight 及 bias 更新时的学习率，这里权重的学习率为 solver.prototxt 文件中定义的学习率，bias 的学习率是权重学习率的两倍，这样一般会得到很好的收敛速度。num_output 表示滤波的个数，kernel_size 表示滤波

的大小, stride 表示步长, weight_filter 表示滤波的类型。LeNet 卷积层如下:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1 //weight 学习率
  }
  param {
    lr_mult: 2 //bias 学习率, 一般为 weight 的两倍
  }
  convolution_param {
    num_output: 20 //滤波器个数
    kernel_size: 5
    stride: 1 //步长
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

LeNet Pooling 池化层如下:

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

LeNet 全连接层如下:

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
}
```



```

    }
    param {
      lr_mult: 2
    }
    inner_product_param {
      num_output: 500
      weight_filler {
        type: "xavier"
      }
      bias_filler {
        type: "constant"
      }
    }
  }
}

```

ReLU 激活函数，非线性变化层 $\max(0, x)$ ，一般与 Convolution 卷积层成对出现。LeNet ReLU 层如下：

```

layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}

```

LeNet Softmax 层如下：

```

layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}

```

4.1.2 参数配置

Caffe 的参数配置文件 *.solver.prototxt 定义了网络模型训练过程中需要设置的参数，比如学习率、权重衰减系数、迭代次数、使用 GPU 还是 CPU 等。如下是 LeNet 的参数配置文件：

```

# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"

# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.

```



```

test_iter: 100

# Carry out testing every 500 training iterations.
test_interval: 500

# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005

# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75

# Display every 100 iterations
display: 100

# The maximum number of iterations
max_iter: 10000

# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"

# solver mode: CPU or GPU
solver_mode: GPU
device_id: 0 #在cmdcaffe接口下, GPU序号从0开始, 如果有一个GPU, 则device_id:0

```

训练出的模型输出文件形式为*.caffemodel, 可以拷贝至目标机器上分类、定位和识别。

4.2 Google Protobuf 结构化数据

Google Protocol Buffer^[1] (简称 Protobuf) 是 Google 公司内部的混合语言数据标准, 目前已经正在使用的有超过 48,162 种报文格式定义和超过 12,183 个 .proto 文件。Google Protobuf 主要用于 RPC 系统和持续数据存储系统。

Protocol Buffers 是一种轻便高效的结构化数据存储格式, 可以用于结构化数据的序列化, 或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通信协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 三种语言的 API。

Protocol Buffers 支持的基本数据类型如表 4-1 所示。

表 4-1 Protocol Buffers 基本数据类型

.proto Type	Notes	C++ Type	Java Type	Python Type
double		double	double	float
float		float	float	float
int32	使用可变长编码, 对于负数比较低效。若负数多, 使用 sint32	int32	int	int
int64	使用可变长编码, 对于负数比较低效。若负数多, 使用 sint64	int64	long	int/long
uint32	使用可变长编码	uint32	int	int/long
uint64	使用可变长编码	uint64	long	int/long
sint32	使用可变长编码, 编码负数比 int32 高效	int32	int	int
sint64	使用可变长编码, 编码负数比 int64 高效	int64	long	int/long
fixed32	恒定四个变长编码。如果数值总大于 2^{28} , 则此类型比 uint32 更高效	uint32	int	Int
fixed64	恒定四个变长编码。如果数值总大于 2^{56} , 则此类型比 uint64 更高效	uint64	long	int/long
sfixed32	恒定四个字节	int32	int	int
sfixed64	恒定八个字节	int64	long	int/long
bool		bool	boolean	Boolean
string	UTF-8 或 7-bit ASCII text	string	String	str/unicode
bytes	包含任意数顺序的字节	string	ByteString	str

Caffe 的网络模型利用 Protocol Buffer (prototxt) 语言定义后存放在 caffe.proto 文件中, 其优势是:

- (1) 当按序排列时, 二进制字符串尺寸最小, 序列化比较高效;
- (2) 文本格式易读, 与二进制版本兼容;
- (3) 多语言接口, 如 C++ 和 Python。

以上优势使得 Caffe 模型具有很强的灵活性与扩展性。

4.3 Caffe 数据库

Caffe 的数据层支持三种格式的数据库^[2]输入：LevelDB、LMDB 和 HDF5。原始图片文件可以通过 `convert_imageset.cpp` 转换成 Caffe 框架支持的数据库文件格式，此文件源代码在 `tools` 文件夹下，生成的可执行文件在 `build/tools` 目录下，使用命令说明如下：

```
convert_imageset [FLAGS] ROOTFOLDER/ LISTFILE DB_NAME
```

此工具有四个参数，如下所述。

- **FLAGS:** 图片参数组，主要包括 `-gray` 是否灰度图片，`-shuffle` 是否随机打乱等；
- **ROOTFOLDER/:** 图片存放的绝对路径，从 Linux 系统根目录开始；
- **LISTFILE:** 图片文件列表清单，一般为一个 `txt` 文件，一行一张图片；
- **DB_NAME:** 最终生成的数据库文件存放目录。

4.3.1 LevelDB

LevelDB 是 Google 实现的一个非常高效的 Key-Value 数据库。它是单进程的服务，性能非常高。

LevelDB 的特点：

- (1) 持久化存储的 KV (Key-Value) 系统，大部分数据存储在磁盘上。
- (2) 以 Key 值有序存储记录数据。
- (3) 接口简单，基本操作有写、读和删除记录等。
- (4) 支持数据快照功能，读取操作有受写操作影响。
- (5) 支持数据压缩操作。

4.3.2 LMDB

LMDB 是一个超级快、超级小的 Key-Value 数据存储服务,由 OpenLDAP 项目的 Symas 开发,使用内存映射文件,其读取性能和内存数据库一样,但大小受限于虚拟地址空间的大小。

LMDB 的主要特性有:

- (1) 基于文件映射 IO (mmap)。
- (2) 基于 B+树的 Key-Value 接口。
- (3) 基于 MVCC (Multi Version Concurrent Control) 的事务处理。
- (4) 类 dbb (Berkeley db) 的 API。

4.3.3 HDF5

HDF (Hierarchical Data File) 是美国国家高级计算应用中心 (NCSA) 为了满足各种领域研究需求而研制的一种能高效存储和分发科学数据的新型数据格式。它可以存储不同类型的图像和数码数据的文件格式,并且可以在不同类型的机器上传输,同时还有统一处理这种文件格式的函数库。HDF5 于 1998 年推出,相比以前的 HDF 文件,是一种全新的文件格式。HDF5 是用于存储科学数据的一种文件格式和库文件。

HDF5 是分层式数据管理结构。HDF5 不但能处理更多的对象,存储更大的文件,支持并行 I/O,线程和具备现代操作系统与应用程序所要求的其他特性,而且数据模型变得更简单,概括性更强。

HDF5 只有两种基本结构:组 (group) 和数据集 (dataset) 组包含 0 个或多个数据集。

4.4 Caffe Net

Caffe Net 表示一个完整的 CNN 模型,是由不同的 layers 组成的有向无环图 (DAG)。

一个典型的 Net 从数据层开始，从外部载入数据，最后在 Loss 层计算目标任务，比如分类和重构^[2]。

Caffe 模型是一个端到端的机器学习引擎，通过合成各层的输出来计算梯度函数，同时通过合成各层的向后传播过程来计算损失函数的梯度，从而完成学习任务。

Net 是由一系列层和它们之间的相互连接构成的，用的是一种文本建模语言。一个简单的逻辑回归分类器的定义如图 4-2 所示。

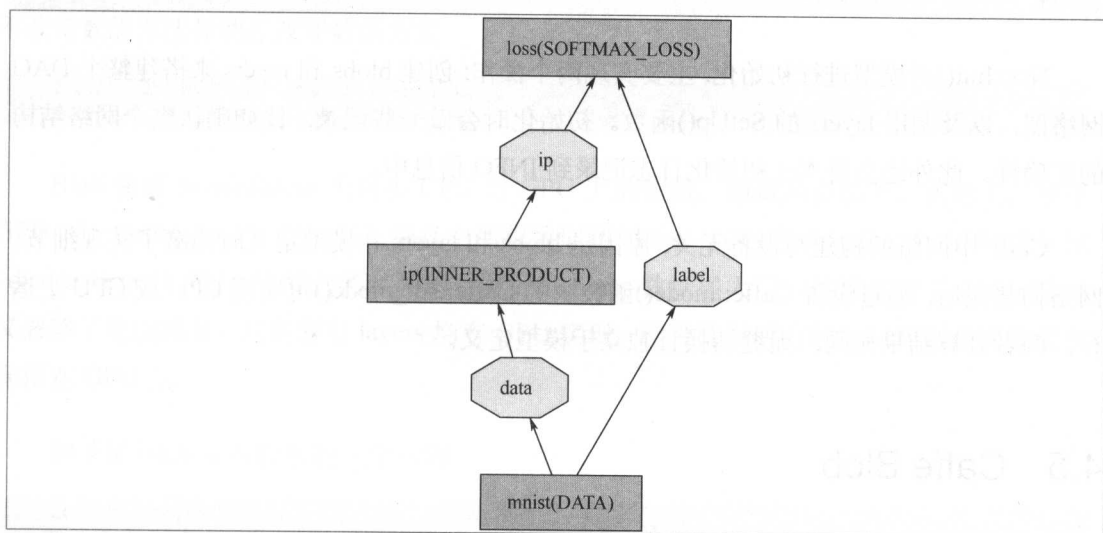


图 4-2 一个简单的逻辑回归分类器的网络模型

其网络模型定义如下：

```

name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"

```

```

    type: "InnerProduct"
    bottom: "data"
    top: "ip"
    inner_product_param {
      num_output: 2
    }
  }
  layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip"
    bottom: "label"
    top: "loss"
  }
}

```

Net::Init()对模型进行初始化, 主要实现两个操作: 创建 blobs 和 layers 来搭建整个 DAG 网络图, 以及调用 layers 的 SetUp()函数。初始化时会做一些记录, 比如确认整个网络结构的正确性。此外还会将 Net 初始化日志记录到 INFO 信息中。

Caffe 中网络的构建与设备无关, 原因是 blobs 和 layers 在模型定义时隐藏了实现细节。网络构建完后, 通过设置 Caffe::mode()函数中的 Caffe::set_mode()可实现 CPU 或 GPU 上运行, 两者计算结果相同, 无缝切换且独立于模型定义。

4.5 Caffe Blob

Caffe 使用 Blob 结构来存储、交换和处理网络中正向和反向迭代的数据和导数信息。Blob 是 Caffe 的标准数组结构, 提供了 Caffe 的统一内存接口。

Blob 是 Caffe 中处理和传递实际数据的数据封装包, 并且在 CPU 与 GPU 之间具有同步处理能力。Blob 还可以根据 CPU 主机到 GPU 设备的同步需要, 屏蔽 CPU/GPU 混合运算在计算上的开销。主机和设备上的内存按需求分配, 以提高内存的使用效率。

一般对批量图像数据, Blob 的维度为图像数量 $N \times$ 通道数 $K \times$ 图像高度 $H \times$ 图像宽度 W 。Blob 按行为主存储, 四维 Blob 中, 坐标为 (n, k, h, w) 的值的物理位置为 $((n \times K + k) \times H + h) \times W + w$ 。例如在 96 个空间的卷积层维度为 11×11 , 输入为 3 通道的滤波器, 则 Blob 的维度为 $96 \times 3 \times 11 \times 11$ 。若一个全连接层 (内积层) 的输入和输出分别是 1024

维和 1000 维，则参数 Blob 的维度等于 $1000 \times 1024^{[2]}$ 。

虽然 Caffe 在图像应用中 Blob 都是四维坐标，对于非图像应用也完全可以正常使用。

Blob 的实现细节。由于 Blob 中最重要的是 values 和 gradients 两类数据，所以 blobs 存储单元对应 data 和 diff 两个数据节点，前者是网络中传递的普通数据，后者是通过网络计算得到的梯度。

Blob 数据既可存储在 CPU 上，也可存储在 GPU 上，因此有两种数据访问方法：静态不改变数值方法和动态改变数值方式。

```
Const Dtype* cpu_data() const;
Dtype* mutable_cpu_data();
```

Blob 使用 SyncedMem 类同步 CPU 与 GPU 上的数值，隐藏同步细节。实际上，使用 GPU 时，Caffe 中 CPU 代码先从磁盘中加载数据到 blobs，同时请求分配一个 GPU 设备核 (device kernel) 进行 GPU 计算，再将计算好的 blobs 数据送入层，这样实现了高效运算，又忽略了底层细节。只要所有 layers 均有 GPU 实现，这种情况所有的中间数据和梯度都会保留在 GPU 上。

如下是 Blob 复制数据的一个示例：

```
Const Dtype* foo;
Dtype* bar;
foo = blob.gpu_data(); //数据从 CPU 复制到 GPU
foo = blob.cpu_data(); //没有数据复制，两者都有最新的内容
bar = blob.mutable_gpu_data(); //没有数据复制
// ... 其他操作...
bar = blob.mutable_gpu_data(); //仍在 GPU，没有数据复制
foo = blob.cpu_data(); //由于 GPU 修改了解数值，数据从 GPU 复制到 CPU
foo = blob.gpu_data(); //没有数据复制，两者都有最新的内容
bar = blob.mutable_cpu_data(); //依旧没有数据复制
bar = blob.mutable_gpu_data(); //数据从 CPU 复制到 GPU
bar = blob.mutable_cpu_data(); //数据从 GPU 复制到 CPU
```


4.6 Caffe Layer

Layer 是 Caffe 模型的本质内容和执行计算的基本单元^[2], 可进行多种运算, 如 convolve (卷积)、pool (池化)、innerproduct (内积)、rectified-linear 和 sigmoid 等非线性运算, 还有元素级的数据变换、normalize (归一化)、load data (数据加载)、softmax 和 hinge 等 losses (损失计算)。在 Caffe 的 Layer catalogue 层目录中可以查看所有的操作, 也可以查看到绝大部分目前最前沿的深度学习任务的层类型。

一个典型的 Caffe 层结构如图 4-3 所示。一个 Layer 通过 bottom 连接层接收数据, 通过 top 连接层输出数据。每一个 layer 都定义了三种重要的运算: setup(初始化设置)、forward (前向传播)、backward (反向传播)。

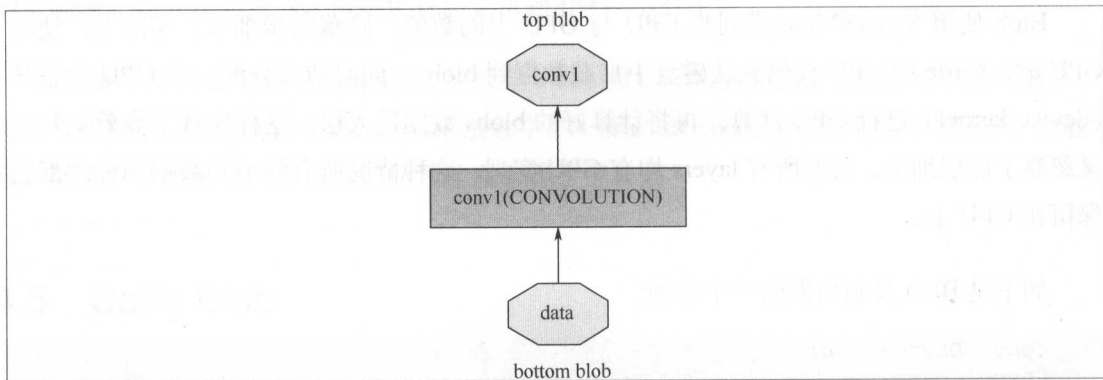


图 4-3 一个典型的 Caffe 层结构

- setup: 在模型初始化时重置 layers 及相互之间的连接;
- forward: 从 bottom 层中接收数据, 计算后将输出送到 top 层;
- backward: 给定 top 层的输出梯度, 计算其输入的梯度, 并传递到 bottom 层。一个有参数的 layer 需要计算相对于各个参数的梯度值并存储在内部。

总之, Layer 承担了网络的两个核心操作: forward pass (前向传播) 接收输入并计算输出; backward pass (反向传播) 接收输出梯度, 计算相对于参数和输入梯度并反向传播给前面的层。这一过程组成了每个 layer 的前向和反向通道。

Caffe 网络的定义和代码实现具有高度的模块化，因此自定义 layer 相对容易，一般只要定义好 layer 的 setup（初始化设置）、forward（前向通道）和 backward（反向通道），就可以将此 layer 加入到 Caffe 网络中。下面我们介绍了一些主要的 Caffe 层类型。

4.6.1 Data Layers

Caffe 的数据层 Data Layer 处于网络的最底层，数据可以从高效率的数据库中读取，也可以直接的内存中读取，或者从硬盘中存储的文件读取。

Data Layer 需要对数据进行预处理操作，常见的操作有减均值、尺度变换、随机裁剪或者镜像等。这些操作可以通过设定参数 Transformation Parameter 来实现。

1. 数据库

- 层类型: Data

- 参数

source	数据库文件的路径	必填
batch_size	网络单次输入数据的数量	必填
rank_skip	跳过开始的 $\text{rank_skip} * \text{rand}(0, 1)$ 个数据，通常在异步随机梯度下降里使用	可选
backend[default LEVELDB]	选择使用 LevelDB 还是 LMDB，默认 LevelDB	可选

如下示例:

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
```

```

    batch_size: 64
    backend: LMDB
  }
}

```

2. 内存数据

- 层类型: MemoryData

- 参数:

batch_size	网络单次输入数据的数量	必填
channels	通道数	必填
height	图像高度	必填
width	图像宽度	必填

说明: memory data 直接从内存中读取数据而不是拷贝, 需要调用 MemoryDataLayer::Reset(C++) 或者 Net.set_input_arrays(Python) 来指定数据来源, 每次读取一个大小为 batch_sized 的数据块。

示例如下:

```

layer {
  top: "data"
  top: "label"
  name: "memory_data"
  type: "MemoryData"
  memory_data_param {
    batch_size: 2
    height: 100
    width: 100
    channels: 1
  }
  transform_param {
    scale: 0.0078125
    mean_file: "mean.proto"
    mirror: false
  }
}

```

3. HDF5数据

● 层类型: HDF5Data

● 参数:

source	读取的文件路径和文件名	必填
batch_size	网络单次输入数据的数量	必填

示例如下:

```
layer {
  name: "data"
  type: "HDF5Data"
  top: "data"
  top: "label"
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
}
```

4. 图像数据Images

● 层类型: ImagesData

● 参数:

source	text 文件的路径名, 此文件的每一行存储一张图片的路径名和对应的标签	必填
batch_size	每一次处理的数据个数, 即图片数	必填
rand_skip	跳过开始的 rand_skip*rand(0, 1)个数据, 通常在异步随机梯度下降里使用	可选
shuffle[default: false]	是否随机打乱图片顺序, 默认为 false	可选
new_height	根据设置的值, 输入的图片将会被调整成给定的高度	可选
new_width	根据设置的值, 输入的图片将会被调整成给定的宽度	可选

示例如下:

```
layer {
  name: "data"
```

```

type: "ImageData"
top: "data"
top: "label"
transform_param {
  mirror: false
  crop_size: 227
  mean_file: "data/ilsrvrc12/imagenet_mean.binaryproto"
}
image_data_param {
  source: "examples/_temp/file_list.txt"
  batch_size: 50
  new_height: 256
  new_width: 256
}
}

```

5. 窗口Windows

- 层类型: WindowsData
- 参数:

source	文件的路径和文件名	必填
batch_size	网络单次输入数据的数量	必填

示例如下:

```

layer {
  name: "data"
  type: "WindowData"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsrvrc12/imagenet_mean.binaryproto"
  }
  window_data_param {
    source:
"examples/finetune_pascal_detection/window_file_2007_trainval.txt"
    batch_size: 128
    fg_threshold: 0.5
    bg_threshold: 0.5
  }
}

```

```

fg_fraction: 0.25
context_pad: 16
crop_mode: "warp"
}
}

```

6. Dummy

- 层类型: DummyData

Dummy 层用于开发与调式。

4.6.2 Convolution Layers

- 层类型: Convolution
- CPU 实现代码: `src/caffe/layers/convolution_layer.cpp`
- CUDA GPU 实现代码: `src/caffe/layers/convolution_layer.cu`
- 参数:

num_output	指定卷积核的数量	必填
kernel_size	指定卷积核的高度和宽度	必填
weight_filler	指定参数的初始化方案	可选
bias_term	指定是否给卷积输出添加偏置项, 默认 true	可选
pad	指定在输入图像周围补 0 的像素个数, 默认 0	可选
stride	指定卷积核在输入图像上滑动的步长, 默认 1	可选
group	指定分组卷积操作的组数, 默认 1	可选

- 输入: $n \times c_i \times h_i \times w_i$
- 输出: $n \times c_o \times h_o \times w_o$, 其中 $h_o = (h_i + 2 \times \text{pad}_h - \text{kernel}_h) / \text{stride}_h + 1$;
- 示例: `models/bvlc_reference_caffenet/train_val.prototxt`

```
layer {
```



```

name: "conv1"
type: "convolution"
bottom: "data"
top: "conv1"
# 卷积核的局部学习率和权值衰减因子
param {lr_mult: 1 decay_mult: 1}
# 偏置项的局部学习率和权值衰减因子
param {lr_mult: 2 decay_mult: 0}
convolution_param {
  num_output: 96      # 学习 96 组卷积核
  kernel_size: 11     # 卷积核大小为 11*11
  stride: 4           # 卷积核滑动步长为 4
  weight_filler{
    type: "gaussian"  # 使用高斯分布随机初始化卷积核
    std: 0.01         # 高斯分布的标准差为 0.01 (默认均值:0)
  }
  bias_filler{
    type: "constant"  # 使用常数 0 初始化偏置项 0
    value: 0 }
  }
}

```

Convolution 层使用一系列可训练的卷积核对输入图像进行卷积操作，每组卷积核生成输出图像中的一个特征图。

4.6.3 Pooling Layers

- 层类型: Pooling
- CPU 实现代码: `src/caffe/layers/pooling_layer.cpp`
- CUDA GPU 实现代码: `src/caffe/layers/pooling_layer.cu`
- 参数:

kernel_size	指定池化窗口的高度和宽度	必填
pool	指池化方法，有三种方法：最大值池化，均值池化，随机池化	可选
pad	指定在输入图像周围补 0 的像素个数，默认 0	可选
stride	指定池化窗口在输入数据上滑动的步长，默认 1	可选

- 输入: $n \times c \times h_i \times w_i$

- 输出: $n*c*h_o*w_o$, h_o 和 w_o 的计算方法与卷积层相同;
- 示例: models/bvlc_reference_caffenet/train_val.prototxt

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX          # 最大值池化方法
    kernel_size: 3      # 池化窗口大小为 3*3
    stride: 2           # 池化窗口在输入图像上滑动的步长为 2
  }
}
```

4.6.4 InnerProduct Layers

- 层类型: InnerProduct (全连接层或内积层)
- CPU 实现代码: src/caffe/layers/inner_product_layer.cpp
- CUDA GPU 实现代码: src/caffe/layers/inner_product_layer.cu
- 参数:

num_output	全连接层的输出节点或滤波器的个数	必填
weight_filler	指定参数的初始化方案, 默认类型为 constant, 值为 0	必填
bias_filler	指定滤波器的偏置类型, 默认类型为 constant, 值为 0	可选
bias_term	指定是否给滤波器添加并训练偏置项, 默认为 True	可选

- 输入: $n*c_i*h_i*w_i$
- 输出: $n*c_o*1*1$
- 示例:

```
layer {
  name: "fc8"
  type: "InnerProduct"
```

```

# learning rate and decay multipliers for the weights
param { lr_mult: 1 decay_mult: 1 }
# learning rate and decay multipliers for the biases
param { lr_mult: 2 decay_mult: 0 }
inner_product_param {
  num_output: 1000
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
bottom: "fc7"
top: "fc8"
}

```

4.6.5 ReLU Layers

- 层类型: ReLU
- CPU 实现代码: src/caffe/layers/relu_layer.cpp
- CUDA GPU 实现代码: src/caffe/layers/relu_layer.cu
- 参数:

negative_slope	设置激活函数在负数部分的斜率。输入数据小于零的部分乘以这个因子，斜率为 0 时，小于零的部分完全滤掉。默认值为 0	可选
----------------	---	----

- 示例: models/bvlc_reference_caffenet/train_val.prototxt

```

layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}

```

给定一个输入数据，当 $x > 0$ 时，ReLU 的输出为 x ，当 $x \leq 0$ 时，输出为 $\text{negative_slope} * x$ 。当 negative_slope 未指定时，等同于标准 ReLU 函数 $\max(x, 0)$ 。

4.6.6 Sigmoid Layers

- 层类型: Sigmoid
- CPU 实现代码: src/caffe/layers/sigmoid_layer.cpp
- CUDA GPU 实现代码: src/caffe/layers/sigmoid_layer.cu
- 参数: 无
- 示例: examples/mnist/mnist_autoencoder.prototxt

```
layer {
  name: "encode1neuron"
  bottom: "encode1"
  top: "encode1neuron"
  type: "Sigmoid"
}
```

Sigmoid 层使用 $\text{sigmoid}(x)$ 函数计算每个输入数据的输出。

4.6.7 LRN Layers

- 层类型: LRN (Local Response Normalization, 局部响应值归一化)
- CPU 实现代码: src/caffe/layers/lrn_layer.cpp
- CUDA GPU 实现代码: src/caffe/layers/lrn_layer.cu
- 参数:

local_size	跨通道时的归一化, 指明参与求和的通道数; 通道内的规范化时, 指明参与求和的方形区域的边长。默认值 5	可选
alpha	尺度参数, 默认值为 1	可选
beta	指数参数, 默认值为 5	可选
norm_region	指定在通道之间进行规范化 (ACROSS_CHANNELS) 还是在通道内规范化 (WITHIN_CHANNEL), 默认值为 ACROSS_CHANNELS	可选

LRN 层通过对输入数的局部归一操作, 执行了一种“侧抑制”的机制。在 ACROSS_

CHANNELS 模式下, 局部区域沿着临近通道延伸, 没有空间扩展。在 WITHIN_CHANNEL 模式下, 局部区域在各自通道内部的图像平面上延伸。

4.6.8 Dropout Layers

- 层类型: Dropout
- CPU 实现代码: src/caffe/layers/dropout_layer.cpp
- CUDA GPU 实现代码: src/caffe/layers/dropout_layer.cu
- 参数:

dropout_ratio	过拟合丢弃数据的概率, 默认 0.5	可选
---------------	--------------------	----

- 示例: models/bvlc_reference_caffenet/train_val.prototxt

```
layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "ip1"
  dropout_param {
    dropout_ratio: 0.5    # 过拟合丢弃率 0.5
  }
}
```

4.6.9 SoftmaxWithLoss Layers

- 层类型: SoftmaxWithLoss
- CPU 实现代码: src/caffe/layers/softmax_loss_layer.cpp
- CUDA GPU 实现代码: src/caffe/layers/softmax_loss_layer.cu
- 参数: 无
- 示例: models/bvlc_reference_caffenet/train_val.prototxt

```

layer {
  name: "loss"
  type: " SoftmaxWithLoss "
  bottom: "ip1"
  bottom: "label"
  top: "loss"
}

```

Softmax_loss 层没有参数项，它封装了一个 softmax 层，即 softmax_loss=softmax+loss regression。此层输出的是 loss 值。

4.6.10 Softmax Layers

- 层类型: Softmax
- CPU 实现代码: src/caffe/layers/softmax_layer.cpp
- CUDA GPU 实现代码: src/caffe/layers/softmax_layer.cu
- 参数: 无
- Softmax 层没有参数项，输出的是似然值。

4.6.11 Accuracy Layers

- 层类型: Accuracy
- CPU 实现代码: src/caffe/layers/accuracy_layer.cpp
- 参数:

phase: include	默认 test 阶段输出分类精确度	必填
----------------	-------------------	----

- 示例: models/bvlc_reference_caffenet/train_val.prototxt

```

layer {
  name: " accuracy "
  type: " Accuracy "
  bottom: "ip2"
  bottom: "label"
}

```

```
top: "accuracy"  
include {  
  phase: TEST  
}
```

4.7 Caffe Solver

Caffe Solver 通过协调 Net 的前向推断计算和反向梯度计算对参数进行更新，从而达到减小 loss 的目的。Caffe 模型的学习分为两个部分：Solver 优化、更新参数，以及 Net 计算 loss 和 gradient。

Caffe 支持 Solvers 有：

- Stochastic Gradient Descent (SGD) 随机梯度下降
- AdaDelta
- Adaptive Gradient (AdaGrad) 自适应梯度
- Adam
- Nesterov's Accelerated Gradient (Nesterov)
- RMSprop

Caffe Solver 的主要功能包括：

- 优化过程记录、创建训练网络和测试网络；
- 通过 forward 和 backward 来迭代优化和更新参数；
- 周期性地用测试网络评估模型性能；
- 在优化过程中记录模型和 Solver 状态的快照。

每一次迭代完成以下功能:

- 调用 Net 的前向过程计算输出和 loss;
- 调用 Net 的后向过程计算梯度;
- 根据 Solver 方法, 利用梯度更新参数;
- 根据学习率, 历史数据和求解方法更新 Solver 状态, 使权重从初始化状态逐步更新到最终的学习状态。

Solver 方法

Solver 方法是计算最小化损失 (loss) 值。给定一个数据集 D , 优化的目标是 D 中所有数据损失的均值, 即平均损失, 取得最小值。

$$L(W) = \frac{1}{|D|} \sum_i f_w(X^{(i)}) + \lambda_r(W)$$

其中 $f_w(X^{(i)})$ 是数据中 $X^{(i)}$ 项的损失, $r(W)$ 是正则项, 权重为 λ 。当 D 数据量很大时, 在每一次迭代中, 采用数据集的一个随机子集来代替, 其数量远小于整个数据集 ($N \ll |D|$)。

$$L(W) \approx \frac{1}{N} \sum_i^N f_w(X^{(i)}) + \lambda_r(W)$$

在前向过程 (forward) 中计算 f_w (即 loss), 在反向过程 (backward) 中计算 ∇f_w (即梯度 gradient)。根据误差度 ∇f_w 、正则项的梯度 $\nabla r(W)$ 以及其他方法的特定项来计算参数更新量 ∇W 。

1. SGD

随机梯度下降 SGD (Stochastic Gradient Descent) 利用负梯度 $\nabla L(W)$ 和上一次权重的更新值 V_t 的线性组合来更新权重 W 。学习率 (Learning Rate) α 是负梯度的权重。动量 μ 是一次更新值的权重。

根据上一次计算的更新值 V_t 和当前权重 W_t 来计算本次的更新值 V_{t+1} 和权重 W_{t+1} :

$$\begin{aligned} V_{t+1} &= \mu V_t - \alpha \nabla L(W_t) \\ W_{t+1} &= W_t + V_{t+1} \end{aligned}$$

学习参数 (α 和 μ) 需要一定的调整才能达到最好的效果^[3]。一般的, 将学习速率 α 初始化为 $0.01=10^{-2}$, 然后在训练中当 loss 达到稳定时, 将 α 除以一个常数 (比如 10), 将这个重复多次。对于动量 μ 设置为 0.9, μ 能使权重的更新更为平缓, 使学习过程更为稳定、快速。

Krizhevsky 在 ILSVRC-2012 竞赛中使用的技巧通过 Caffe 中的 SolverParameter 很容易实现^[4]。具体方法是将下面的代码添加到自定义的 solver prototxt 文件中。

```
base_lr:0.01;           #/ 开始学习速率为:  $\alpha=0.01=1e-2$ 
lr_policy: "step"       # 学习策略: 每个步长迭代之后, 将  $\alpha$  乘以 gamma
gamma: 0.1              # 学习速率变化因子
stepsize:100000         # 每 100K 次迭代, 降低学习速率
max_iter:350000         # 训练的最大迭代次数 350K
momentum:0.9            # 动量 momentum  $\mu=0.9$ 
```

本例中, 动量 μ 设置为常数 0.9。前 100 次迭代时学习速率 (base_lr) $\alpha=0.01=10^{-2}$, 然后将学习速率 α 乘以 gamma, 即在第 100K-200K 次迭代时以学习速率 $\alpha' = \alpha\gamma=(0.01)(0.1)=0.001=10^{-3}$ 进行训练, 之后第 200K-300K 次迭代时学习速率 $\alpha'' = 10^{-4}$, 最后第 301K-350K 次迭代时学习速率为 $\alpha=10^{-5}$ 。

当训练次数达到一定量后, 更新值会扩大到 $\frac{\mu}{1-\mu}$ 倍, 所以如果增加 μ , 相应地需要减少 α 值 (反之亦然)。比如设 $\mu=0.9$, 则更新值会扩大 $\frac{1}{1-0.9}=10$ 倍。若将 μ 扩大为 0.99, 那么更新值会扩大 100 倍, 所以 α 应该除以 10。

以上方法是基于经验的, 在实际训练过程中如果出现了发散环境, 可以通过减小基准学习速率再训练, 并且重复这个过程, 直到找到一个比较合适的学习速率。

2. AdaDelta

AdaDelta 方法是一种“鲁棒的学习率方法”^[5]，同 SGD 一样是一种基于梯度的优化方法。更新方程如下：

$$(v_t)_i = \frac{RMS((v_{t-1})_i)}{RMS(\nabla L(W)_t)_i} (\nabla L(W)_t)_i$$

$$RMS(\nabla L(W)_t)_i = \sqrt{E[g^2] + \varepsilon}$$

$$E[g^2]_t = \delta E[g^2]_{t-1} + (1 - \delta) g_t^2$$

$$(W_{t+1})_i = (W_t)_i - \alpha (v_t)_i$$

3. AdaGrad

自适应性梯度下降方法 (Adaptive Gradient)^[6]与随机梯度下降 (Stochastic Gradient) 一样是基于梯度的优化方法。设更新前信息为 $(\nabla L(W))_{t'}$ ， $t' \in \{1, 2, \dots, t\}$ ，权重 W 中每个元素 i 的更新如下：

$$(W_{t+1})_i = (W_t)_i - \alpha \frac{(\nabla L(W))_i}{\sqrt{\sum_{t'=1}^t (\nabla L(W_{t'}))_i^2}}$$

实际使用中，对于权重 $W \in R^d$ ，自适应梯度 (AdaGrad) 的实现需要 $O(d)$ 额外空间存储历史梯度信息。

4. Adam

Adam 也是一种基于梯度的优化方法。它包含一对自适应时刻估计为量 (adaptive moment estimation) (m_t, μ_t) ，可以看做是 AdaGrad 的一种泛化形式。

$$(m_t)_i = \beta_1 (m_{t-1})_i + (1 - \beta_1) (\nabla L(W_t))_i$$

$$(v_t)_i = \beta_2 (v_{t-1})_i + (1 - \beta_2) (\nabla L(W_t))_i^2$$

$$(W_{t+1})_i = (W_t)_i - \alpha \frac{\sqrt{1 - (\beta_2)_i^t}}{\sqrt{1 - (\beta_1)_i^t}} \frac{(m_t)_i}{\sqrt{(v_t)_i + \varepsilon}}$$

Kingma et al^[7]提出 $\beta_1=0.9$, $\beta_2=0.999$, $\varepsilon=10^{-8}$ 作为默认值, Caffe 中使用 momentum, momentum2, delta 分别代表 β_1 , β_2 和 ε 。

5. NAG

Nesterov^[8]提出加速梯度下降 (Nesterov's accelerated gradient) 是凸优化的一种最优算法, 其收敛速度可以达到 $O(1/t^2)$ 。此方法的权重更新参数与随机梯度下降非常相似:

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t + \mu V_t)$$

$$W_{t+1} = W_t + V_{t+1}$$

与 SGD 不同在于梯度 $\nabla L(W)$ 项中取值不同。

6. RMSprop

Tieleman^[9]提及的 RMSprop 方法同样是一种基于梯度的优化方法。更新方程如下:

$$(v_t)_i = \begin{cases} (v_{t-1})_i + \delta, & (\nabla L(W_t))_i (\nabla L(W_{t-1}))_i > 0 \\ (v_{t-1})_i \cdot (1 - \delta), & \text{else} \end{cases}$$

$$(W_{t+1})_i = (W_t)_i - \alpha (v_t)_i$$

如果梯度更新值产生振动, 则让梯度减少 (即乘以 $(1-\delta)$), 否则增加 δ 。 δ 默认值是 0.02。

参考文献

- [1] <https://developers.google.com/protocol-buffers/>, Google protobuf 开发者网站
- [2] Yangqing Jia, Evan Shelhamer, Caffe Tutorial, <http://caffe.berkeleyvision.org/tutorial/>, 2016.
- [3] L.Bottou. Stochastic Gradient Descent Tricks. Neural Networks: Tricks of the Trade: Springer, 2012
- [4] A.Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural

Networks, Advance in Neural Information Processing System, 2012.

- [5] M. Zeiler AdaDelta: An adaptive Learning Rate Method. arXiv preprint, 2012
- [6] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. The Journal of Machine Learning Research, 2011.
- [7] D.Kingma, J. Ba. Adam: A Method for Stochastic Optimization. International Conference for Learning Representation, 2015.
- [8] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/\sqrt{k})$. Soviet Mathematics Doklady, 1983.
- [9] T. Tieleman, and G. Hinton. RMSProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning. Technical report, 2012

5

第 5 章 LeNet 模型

通过前面的章节，我们已经对 Caffe 深度学习框架有了具体的认识 and 了解。LeNet 作为 Caffe 入门级的网络模型，解读其设计思想能从实践角度，进一步帮助我们理解深度学习的原理。LeNet 是一个用来识别手写数字的最经典的卷积神经网络，是 Yann LeCun 在 1998 年设计并提出的，是早期卷积神经网络中最有代表性的实验系统之一，其论文是 CNN 领域的第一篇经典之作。基于此神经网络模型的手写数字识别系统 LeNet-5 在 20 世纪 90 年代被广泛用于银行手写数字的识别系统。

本章先介绍 LeNet 网络模型并进行解读，然后详细介绍了 LeNet 训练和测试方法，本章的最后给读者提供了一个用 Python 语言实现的样本字库的转换脚本。

5.1 LeNet 模型简介

根据 Yann LeCun 公开发表的论文内容，LeNet-5 卷积神经网络的整体框架如图 5-1 所示。可以看出 LeNet 的网络结构规模比较小，但包含了卷积层、Pooling 层、全连接层，他们构成了现代 CNN 网络的基本组件，后续更复杂的网络模型都离不开这些基本网络层组件，因此学习和研究 LeNet 模块是研究更复杂网络模型的基础。

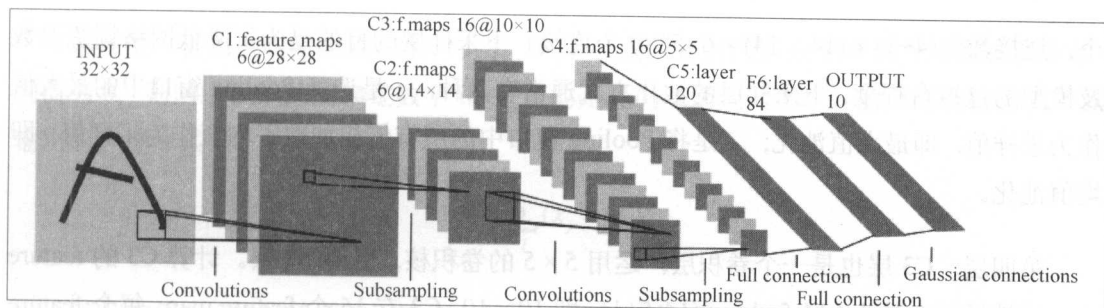


图 5-1 LeNet 网络模型框架

LeNet-5 包含输入层在内共有八层，每一层都包含多个参数（权重）。C 层代表的是卷积层，通过卷积操作，可以使原信号特征增强，并降低噪音。S 层是一个下采样层，利用图像局部相关性的原理，对图像进行子抽样，可以减少数据处理量，同时也可保留一定的有用信息。

5.2 LeNet 模型解读

LeNet 网络模型的具体结构如下：

第一层：输入层是 32×32 大小的图像（Caffe 中 Mnist 数据库为 28×28 ），这样做的原因是希望潜在的明显特征，如笔画断续、角点能够出现在最高层特征监测子感受野的中心。

第二层：C1 层是一个卷积层，6 个特征图（feature map）， 5×5 大小的卷积核，每个 feature map 有 $(32-5+1) \times (32-5+1)$ ，即 28×28 个神经元，每个神经元都与输入层的 5×5 大小的区域相连。故 C1 层共有 $(5 \times 5+1) \times 6=156$ 个训练参数。两层之间的连接数为 $156 \times (28 \times 28)=122304$ 个。通过卷积运算，使原信号特征增强，并且降低噪音，而且不同的卷积核能够提取到图像中的不同特征。

第三层：S2 层一个下采样层，有 6 个 14×14 的特征图（feature map），每个 feature map 中的每个神经元都与 C1 层对应的 feature map 中的 2×2 的区域相连。S2 层中的每个神经元是由这 4 个输入相加，乘以一个训练参数，再加上这个 feature map 的偏置参数，结果通过 sigmoid 函数计算而得。S2 的每一个 feature map 有 14×14 个神经元，参数个数为 $2 \times 6=12$

个, 连接数为 $(4+1) \times (14 \times 14) \times 6=5880$ 个连接。下采样层的目的是为了降低网络训练参数及模型的过拟合程度。下采样层的池化方式通常有两种: 一是选择 Pooling 窗口中的最大值作为采样值, 即最大值池化; 二是将 Pooling 窗口中的所有值相加取平均值作为采样值, 即均值池化。

第四层: C3 层也是一个卷积层, 运用 5×5 的卷积核, 处理 S2 层。计算 C3 的 feature map 的神经元个数为 $(14-5+1) \times (14-5+1)$, 即 10×10 。C3 有 16 个 feature map, 每个 feature map 由上一层的各 feature map 之间的不同组合。

图 5-2 汇总了 C3 层的 feature map 组合情况, 其第 0 个 feature map 是由 S2 层的第 0、1、2 个 feature map 组合得到的, 第 1 个 feature map 是由 S2 层的第 1、2、3 个 feature map 组合得到的, 依此类推。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X	X		X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X	X		X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

图 5-2 LeNet 第三层 feature map 组合

计算出 C3 层的训练参数个数为 $(5 \times 5 \times 3+1) \times 6+(5 \times 5 \times 4+1) \times 9+(5 \times 5 \times 6+1) \times 1=1516$, 因此有 151600 个连接。

第五层: S4 层是一个下采样层, 由 16 个 5×5 大小的 feature map 构成, 每个神经元与 C3 中对应 feature map 的 2×2 大小的区域相连。同理, 计算出 $2 \times 16=32$ 个参数和 2000 个连接。

第六层: C5 层又是一个卷积层, 同样使用 5×5 的卷积核, 每个 feature map 有 $(5-5+1) \times (5-5+1)$, 即 1×1 个神经元, 每个单元都与 S4 层的全部 16 个 feature map 的 5×5 区域全相连。C5 层共有 120 个 feature map, 其参数与连接数都为 48120 个。

第七层: F6 全连接层有 84 个 feature map, 每个 feature map 只有一个神经元与 C5 层全相连, 故有 $(1 \times 1 \times 120+1) \times 84=10164$ 个参数和连接。F6 层计算输入向量和权重向量之间的点积和偏置, 之后将其传递给 sigmoid 函数来计算神经元。

第八层：输出层也是全连接层，共有 10 个节点，分别代表数字 0 到 9，且如果节点 i 的值为 0，则网络识别的结果是数字 i 。采用的是径向基函数（RBF）的网络连接方式，其输出的计算方式是：

$$y_i = \sum_j (x_j - w_{ij})^2$$

RBF 的值由 i 的比特图编码确定。越接近于 0，则越接近于 i 的比特图编码，表示当前网络输入的识别结果是字符 i 。该层有 $84 \times 10 = 840$ 个设定的参数和连接。

5.3 Caffe 环境 LeNet 模型

5.3.1 mnist 实例详解

Caffe 安装包自带了 mnist 的例子。因此安装好 Caffe 后，在 `caffe-master/exmaples/mnist` 目录可以找到 LeNet 模型的具体实现。测试步骤如下：

1. 数据下载

获得 mnist 的数据包，在 caffe 根目录下执行 `./data/mnist/get_mnist.sh` 脚本，脚本内容和注释如下：

```
#!/usr/bin/env sh
# This scripts downloads the mnist data and unzips it.
DIR="$( cd "$(dirname "$0")" ; pwd -P )"
cd $DIR
echo "Downloading..."

# 下载 LeNet 样本库，共四文件
wget --no-check-certificate http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz // 训练样本
wget --no-check-certificate http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz // 训练标签
wget --no-check-certificate http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz // 测试数据
wget --no-check-certificate http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz // 测试标签
```

```

echo "Unzipping..."

# 解压样本库压缩文件
gunzip train-images-idx3-ubyte.gz
gunzip train-labels-idx1-ubyte.gz
gunzip t10k-images-idx3-ubyte.gz
gunzip t10k-labels-idx1-ubyte.gz

# Creation is split out because leveldb sometimes causes segfault
# and needs to be re-created.

echo "Done."

```

get_mnist.sh 脚本先从 <http://yann.lecun.com/exdb/mnist> 下载样本库并进行解压缩，共有四个文件：train-images-idx3-ubyte.gz、train-labels-idx1-ubyte.gz、t10k-images-idx3-ubyte.gz 和 t10k-labels-idx1-ubyte.gz。

2. 生成LMDB

成功解压缩下载的样本库后，然后执行./example/mnist/create_mnist.sh，脚本内容和注释如下：

```

#!/usr/bin/env sh
# This script converts the mnist data into lmdb/leveldb format,
# depending on the value assigned to $BACKEND.

EXAMPLE=examples/mnist
DATA=data/mnist
BUILD=build/examples/mnist

BACKEND="lmdb"

echo "Creating ${BACKEND}..."

# 清空原有目录的内容
rm -rf $EXAMPLE/mnist_train ${BACKEND}
rm -rf $EXAMPLE/mnist_test ${BACKEND}

# 用 build/examples/mnist/convert_mnist_data.bin 命令转换 lmdb 格式
$BUILD/convert_mnist_data.bin $DATA/train-images-idx3-ubyte \
    $DATA/train-labels-idx1-ubyte $EXAMPLE/mnist_train_${BACKEND} --backend=
${BACKEND}
$BUILD/convert_mnist_data.bin $DATA/t10k-images-idx3-ubyte \
    $DATA/t10k-labels-idx1-ubyte $EXAMPLE/mnist_test_${BACKEND} --backend=
${BACKEND}

```

```
echo "Done."
```

create_mnist.sh 脚本先利用 caffe-master/build/examples/mnist/目录下的 convert_mnist_data.bin 工具, 将 mnist data 转化 caffe 可用的 lmdb 格式文件, 然后将生成的 mnist-train-lmdb 和 mnist-test-lmdb 两个文件放在 caffe-master/example/mnist 目录下面。

3. 网络配置

LeNet 网络定义在 ./examples/mnist/lenet_train_test.prototxt 文件中, 只需要注意 source 参数文件路径, 其他参数不需要修改, 文件内容和注释如下:

```
name: "LeNet"
layer {
  name: "mnist"    // 输入层的名称 mnist
  type: "Data"     // 输入层的类型为 Data 层
  top: "data"      // 本层下一层连接 data 层和 label blob 空间
  top: "label"
  include {
    phase: TRAIN   // 训练阶段
  }
  transform_param {
    scale: 0.00390625 // 输入图片像素归一到 [0, 1], 1 除以 256 为 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
                                // 从 mnist_train_lmdb 中读入数据
    batch_size: 64             // batch 大小为 64, 一次训练 64 条数据
    backend: LMDB
  }
}
layer {
  name: "mnist"    // 输入层的名称 mnist
  type: "Data"     // 输入层的类型为 Data 层
  top: "data"      // 本层下一层连接 data 层和 label blob 空间
  top: "label"
  include {
    phase: TEST     // 测试阶段
  }
  transform_param {
    scale: 0.00390625 // 输入图片像素归一到 [0, 1], 1 除以 256 为 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
                                // 从 mnist_test_lmdb 中读入数据
```

```

    batch_size: 100 // batch 大小为 100, 一次测试 100 条数据
    backend: LMDB
  }
}
layer {
  name: "conv1" // 卷积层名称 conv1
  type: "Convolution" // 层类型为卷积层
  bottom: "data" // 本层使用上一层的 data, 生成下一层 conv1 的 blob
  top: "conv1"
  param {
    lr_mult: 1 // 权重参数 w 的学习率倍数
  }
  param {
    lr_mult: 2 // 偏置参数 b 的学习率倍数
  }
  convolution_param {
    num_output: 20 // 输出单元数 20
    kernel_size: 5 // 卷积核大小为 5*5
    stride: 1 // 步长为 1
    weight_filler { // 允许用随机值初始化权重和偏置值
      type: "xavier" // 使用 xavier 算法自动确定基于输入-输出神经元数量的初始规模
    }
    bias_filler {
      type: "constant" // 偏置值初始化为常数, 默认为 0
    }
  }
}
}
layer {
  name: "pool1" // 层名称为 pool1
  type: "Pooling" // 层类型为 pooling
  bottom: "conv1" // 本层的上一层是 conv1, 生成下一层 pool1 的 blob
  top: "pool1"
  pooling_param { // pooling 层的参数
    pool: MAX // pooling 的方式是 MAX
    kernel_size: 2 // pooling 核是 2*2
    stride: 2 // pooling 步长是 2
  }
}
}
layer { // 第二个卷积层, 同第一个卷积层相同, 只是卷积核为 50;
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
}

```



```

convolution_param {
  num_output: 50
  kernel_size: 5
  stride: 1
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}
}
layer {                                     //第二个 pooling 层, 与第一个 pooling 层相同
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {                                     // 全连接层
  name: "ip1"                               // 全连接层名称 ip1
  type: "InnerProduct"                     // 层类型为全连接层
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {                    // 全连接层的参数
    num_output: 500                        // 输出 500 个节点
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {                                     // ReLU 层
  name: "relu1"                             // 层名称为 relu1
  type: "ReLU"                             // 层类型为 ReLU
  bottom: "ip1"
  top: "ip1"
}

```

```

layer {                                // 第二个全连接层
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10           // 输出 10 个单元
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {                                // Loss 层, softmax_loss 层实现 softmax 和多项 Logistic 损失,
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}

```

4. 训练网络

运行 `./example/mnist/train_lenet.sh`, 其内容如下:

```

#!/usr/bin/env sh

./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt

```

执行此脚本时, 实际运行的是 `lenet_solver.prototxt` 中的定义, 其内容和注释如下:

```

# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"           //网络具体定义
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100      // test 迭代次数, 若 batch_size =100, 则 100 张图一批, 训练 100
次, 可覆盖 10000 张图
# Carry out testing every 500 training iterations.
test_interval: 500  // 训练迭代 500 次, 测试一次
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01       // 网络参数: 学习率, 动量, 权重的衰减
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy // 学习策略: 有固定学习率和每步递减学习率
lr_policy: "inv"    // 当前使用递减学习率
gamma: 0.0001
power: 0.75
# Display every 100 iterations // 每迭代 100 次显示一次
display: 100
# The maximum number of iterations // 最大迭代次数
max_iter: 10000
# snapshot intermediate results // 每 5000 次迭代存储一次数据
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: CPU    // 本例用 CPU 训练

```

训练 LeNet 时, 部分打印输出和注解如下:

```

I0816 11:50:49.272727 2307 caffe.cpp:178] Use CPU.
I0816 11:50:49.273232 2307 solver.cpp:48] Initializing solver from
parameters: //初始化参数...
test_iter: 100
test_interval: 500
base_lr: 0.01
display: 100
max_iter: 10000
lr_policy: "inv"
gamma: 0.0001
power: 0.75
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
solver_mode: CPU    // CPU 训练
net: "examples/mnist/lenet_train_test.prototxt"
I0816 11:50:49.295636 2307 solver.cpp:91] Creating training net from net
file: examples/mnist/lenet_train_test.prototxt
I0816 11:50:49.295862 2307 net.cpp:313] The NetState phase (0) differed from

```



```

the phase (1) specified by a rule in layer mnist
I0816 11:50:49.295941 2307 net.cpp:313] The NetState phase (0) differed from
the phase (1) specified by a rule in layer accuracy
I0816 11:50:49.296047 2307 net.cpp:49] Initializing net from parameters:
..... // 省略部分打印
I0816 11:56:20.368371 2307 sgd_solver.cpp:106] Iteration 8400, lr =
0.00632975 //8400 次迭代
I0816 11:56:23.873293 2307 solver.cpp:337] Iteration 8500, Testing net (#0)
I0816 11:56:26.189328 2307 solver.cpp:404] Test net output #0: accuracy
= 0.9905
I0816 11:56:26.189512 2307 solver.cpp:404] Test net output #1: loss =
0.0305883 (* 1 = 0.0305883 loss)
I0816 11:56:26.222558 2307 solver.cpp:228] Iteration 8500, loss = 0.00713543
I0816 11:56:26.222719 2307 solver.cpp:244] Train net output #0: loss =
0.00713541 (* 1 = 0.00713541 loss)
I0816 11:56:26.222792 2307 sgd_solver.cpp:106] Iteration 8500, lr=0.00630407
I0816 11:56:29.759721 2307 solver.cpp:228] Iteration 8600, loss = 0.00107963
I0816 11:56:29.760012 2307 solver.cpp:244] Train net output #0: loss =
0.00107961 (* 1 = 0.00107961 loss)
I0816 11:56:29.760085 2307 sgd_solver.cpp:106] Iteration 8600, lr=0.00627864
I0816 11:56:33.293192 2307 solver.cpp:228] Iteration 8700, loss = 0.00194229
I0816 11:56:33.293344 2307 solver.cpp:244] Train net output #0: loss =
0.00194228 (* 1 = 0.00194228 loss)
I0816 11:56:33.293416 2307 sgd_solver.cpp:106] Iteration 8700, lr=0.00625344
I0816 11:56:36.794823 2307 solver.cpp:228] Iteration 8800, loss = 0.00132375
I0816 11:56:36.795073 2307 solver.cpp:244] Train net output #0: loss =
0.00132374 (* 1 = 0.00132374 loss)
I0816 11:56:36.795156 2307 sgd_solver.cpp:106] Iteration 8800, lr=0.00622847
I0816 11:56:40.330515 2307 solver.cpp:228] Iteration 8900, loss = 0.00169368
I0816 11:56:40.331105 2307 solver.cpp:244] Train net output #0: loss =
0.00169366 (* 1 = 0.00169366 loss)
I0816 11:56:40.331179 2307 sgd_solver.cpp:106] Iteration 8900, lr=0.00620374
I0816 11:56:43.831454 2307 solver.cpp:337] Iteration 9000, Testing net (#0)
I0816 11:56:46.111595 2307 solver.cpp:404] Test net output #0: accuracy =
0.9901
I0816 11:56:46.113252 2307 solver.cpp:404] Test net output #1: loss =
0.0298207 (* 1 = 0.0298207 loss)
I0816 11:56:46.145944 2307 solver.cpp:228] Iteration 9000, loss = 0.0168464
I0816 11:56:46.146103 2307 solver.cpp:244] Train net output #0: loss =
0.0168464 (* 1 = 0.0168464 loss)
I0816 11:56:46.146173 2307 sgd_solver.cpp:106] Iteration 9000, lr=0.00617924
I0816 11:56:49.681800 2307 solver.cpp:228] Iteration 9100, loss = 0.0093805
I0816 11:56:49.681965 2307 solver.cpp:244] Train net output #0: loss =
0.00938048 (* 1 = 0.00938048 loss)
I0816 11:56:49.682040 2307 sgd_solver.cpp:106] Iteration 9100, lr=0.00615496
I0816 11:56:53.212987 2307 solver.cpp:228] Iteration 9200, loss = 0.00349179
I0816 11:56:53.213660 2307 solver.cpp:244] Train net output #0: loss =
0.00349177 (* 1 = 0.00349177 loss)
I0816 11:56:53.213734 2307 sgd_solver.cpp:106] Iteration 9200, lr=0.0061309
I0816 11:56:56.746960 2307 solver.cpp:228] Iteration 9300, loss = 0.00660118
I0816 11:56:56.747953 2307 solver.cpp:244] Train net output #0: loss =

```

```

0.00660115 (* 1 = 0.00660115 loss)
I0816 11:56:56.748030 2307 sgd_solver.cpp:106] Iteration 9300, lr=0.00610706
I0816 11:57:00.374629 2307 solver.cpp:228] Iteration 9400, loss = 0.0217692
I0816 11:57:00.375195 2307 solver.cpp:244] Train net output #0: loss =
0.0217692 (* 1 = 0.0217692 loss)
I0816 11:57:00.375269 2307 sgd_solver.cpp:106] Iteration 9400, lr=0.00608343
I0816 11:57:03.870839 2307 solver.cpp:337] Iteration 9500, Testing net (#0)
I0816 11:57:06.189862 2307 solver.cpp:404] Test net output #0: accuracy =
0.9892
I0816 11:57:06.190023 2307 solver.cpp:404] Test net output #1: loss =
0.0344825 (* 1 = 0.0344825 loss)
I0816 11:57:06.220799 2307 solver.cpp:228] Iteration 9500, loss = 0.00468386
I0816 11:57:06.223397 2307 solver.cpp:244] Train net output #0: loss =
0.00468382 (* 1 = 0.00468382 loss)
I0816 11:57:06.223474 2307 sgd_solver.cpp:106] Iteration 9500, lr=0.00606002
I0816 11:57:09.758777 2307 solver.cpp:228] Iteration 9600, loss = 0.00251893
I0816 11:57:09.759034 2307 solver.cpp:244] Train net output #0: loss =
0.0025189 (* 1 = 0.0025189 loss)
I0816 11:57:09.759107 2307 sgd_solver.cpp:106] Iteration 9600, lr=0.00603682
I0816 11:57:13.263119 2307 solver.cpp:228] Iteration 9700, loss = 0.00281375
I0816 11:57:13.263284 2307 solver.cpp:244] Train net output #0: loss =
0.00281372 (* 1 = 0.00281372 loss)
I0816 11:57:13.263356 2307 sgd_solver.cpp:106] Iteration 9700, lr=0.00601382
I0816 11:57:16.801949 2307 solver.cpp:228] Iteration 9800, loss = 0.0136321
I0816 11:57:16.802126 2307 solver.cpp:244] Train net output #0: loss =
0.0136321 (* 1 = 0.0136321 loss)
I0816 11:57:16.802196 2307 sgd_solver.cpp:106] Iteration 9800, lr=0.00599102
I0816 11:57:20.337122 2307 solver.cpp:228] Iteration 9900, loss = 0.00555447
I0816 11:57:20.337733 2307 solver.cpp:244] Train net output #0: loss =
0.00555444 (* 1 = 0.00555444 loss)
I0816 11:57:20.337811 2307 sgd_solver.cpp:106] Iteration 9900, lr=0.00596843
I0816 11:57:23.810355 2307 solver.cpp:454] Snapshotting to binary proto file
examples/mnist/lenet_iter_10000.caffemodel // 10000 次迭代时, 输出训练好的 model 文
件
I0816 11:57:23.815733 2307 sgd_solver.cpp:273] Snapshotting solver state to
binary proto file examples/mnist/lenet_iter_10000.solverstate // 1000 次迭代时,
输出 solverstate 文件。
I0816 11:57:23.831202 2307 solver.cpp:317] Iteration 10000, loss = 0.00225965
I0816 11:57:23.832715 2307 solver.cpp:337] Iteration 10000, Testing net (#0)
I0816 11:57:26.145452 2307 solver.cpp:404] Test net output #0: accuracy =
0.9903
I0816 11:57:26.145608 2307 solver.cpp:404] Test net output #1: loss =
0.0285704 (* 1 = 0.0285704 loss)
I0816 11:57:26.145678 2307 solver.cpp:322] Optimization Done.
I0816 11:57:26.145741 2307 caffe.cpp:222] Optimization Done.

```

数据训练结束后, 会生成 lenet_iter_10000.caffemodel、lenet_iter_10000.solverstate、lenet_iter_5000.caffemodel 和 lenet_iter_5000.solverstate 四个文件。

训练的网络结构汇总如表 5-1 所示。

表 5-1 LeNet 模型训练的网络结构

序号	Layer	Layer Type Bottom	Blob Bottom	Blob Top	Blob Shape
1	minst	Data		Data&Label	64 1 28 28 (50176)
2	conv1	Convolution	data	conv1	64 20 24 24 (737280)
3	pool1	Pooling	conv1	pool1	64 20 12 12 (184320)
4	conv2	Convolution	pool1	conv2	64 50 8 8 (204800)
5	pool2	Pooling	conv2	pool2	64 50 4 4 (51200)
6	ip1	InnerProduct	pool2	ip1	64 500 (32000)
7	relu1	ReLU	ip1	ip1(in-place)	64 500 (32000)
8	ip2	InnerProduct	ip1	ip2	64 10 (640)
9	loss	SoftmaxWithLoss	ip2&&label	loss	(1)

注：Top Blob Shap 格式为：BatchSize, ChannelSize, Height, Width (Total Count)。

5. 测试网络

运行 ./example/mnist/test_lenet.sh 脚本，此脚本调用 tools 目录下面的 caffe 命令完成测试，命令参数指定网络定义文件名和训练出来的 caffemodel 文件，如下是测试命令显示的部分内容和注释如下：

```
I0816 14:40:29.466565 2801 caffe.cpp:246] Use CPU.
I0816 14:40:29.467777 2801 net.cpp:313] The NetState phase (1) differed from
the phase (0) specified by a rule in layer minst
I0816 14:40:29.467926 2801 net.cpp:49] Initializing net from parameters:
name: "LeNet"
state {
  phase: TEST
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST // 测试阶段
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
```



```

source: "examples/mnist/mnist_test_lmdb" // 指定测试数据集的路径
batch_size: 100 // 考虑到测试效率, 每个 batch 大小为 100
backend: LMDB
}
}
..... //省略部分打印
I0816 14:40:29.476205 2801 net.cpp:219] mnist does not need backward
computation.
I0816 14:40:29.476217 2801 net.cpp:261] This network produces output accuracy
I0816 14:40:29.476229 2801 net.cpp:261] This network produces output loss
I0816 14:40:29.476248 2801 net.cpp:274] Network initialization done.
I0816 14:40:29.480383 2801 caffe.cpp:252] Running for 50 iterations.
I0816 14:40:29.512894 2801 caffe.cpp:275] Batch 0, accuracy = 1
I0816 14:40:29.512982 2801 caffe.cpp:275] Batch 0, loss = 0.00179199
I0816 14:40:29.534860 2801 caffe.cpp:275] Batch 1, accuracy = 1
I0816 14:40:29.535416 2801 caffe.cpp:275] Batch 1, loss = 0.00503976
I0816 14:40:29.557428 2801 caffe.cpp:275] Batch 2, accuracy = 0.99
I0816 14:40:29.557585 2801 caffe.cpp:275] Batch 2, loss = 0.0337807
I0816 14:40:29.580363 2801 caffe.cpp:275] Batch 3, accuracy = 0.99
I0816 14:40:29.580503 2801 caffe.cpp:275] Batch 3, loss = 0.0250572
I0816 14:40:29.602743 2801 caffe.cpp:275] Batch 4, accuracy = 0.98
I0816 14:40:29.603564 2801 caffe.cpp:275] Batch 4, loss = 0.0877343
I0816 14:40:29.625854 2801 caffe.cpp:275] Batch 5, accuracy = 0.99
I0816 14:40:29.626072 2801 caffe.cpp:275] Batch 5, loss = 0.0281052
I0816 14:40:29.648236 2801 caffe.cpp:275] Batch 6, accuracy = 0.96
I0816 14:40:29.648393 2801 caffe.cpp:275] Batch 6, loss = 0.0716703
I0816 14:40:29.670599 2801 caffe.cpp:275] Batch 7, accuracy = 0.99
I0816 14:40:29.670770 2801 caffe.cpp:275] Batch 7, loss = 0.0283097
I0816 14:40:29.693300 2801 caffe.cpp:275] Batch 8, accuracy = 1
I0816 14:40:29.693974 2801 caffe.cpp:275] Batch 8, loss = 0.00511408
I0816 14:40:29.715543 2801 caffe.cpp:275] Batch 9, accuracy = 0.99
I0816 14:40:29.715653 2801 caffe.cpp:275] Batch 9, loss = 0.0393358
I0816 14:40:29.738306 2801 caffe.cpp:275] Batch 10, accuracy = 0.98
I0816 14:40:29.738453 2801 caffe.cpp:275] Batch 10, loss = 0.057836
I0816 14:40:29.760319 2801 caffe.cpp:275] Batch 11, accuracy = 0.98
I0816 14:40:29.760499 2801 caffe.cpp:275] Batch 11, loss = 0.0488569
I0816 14:40:29.782023 2801 caffe.cpp:275] Batch 12, accuracy = 0.96
I0816 14:40:29.783124 2801 caffe.cpp:275] Batch 12, loss = 0.118516
I0816 14:40:29.806447 2801 caffe.cpp:275] Batch 13, accuracy = 0.98
I0816 14:40:29.806838 2801 caffe.cpp:275] Batch 13, loss = 0.038092
I0816 14:40:29.828631 2801 caffe.cpp:275] Batch 14, accuracy = 1
I0816 14:40:29.829216 2801 caffe.cpp:275] Batch 14, loss = 0.00812573
I0816 14:40:29.851001 2801 caffe.cpp:275] Batch 15, accuracy = 0.99
I0816 14:40:29.851155 2801 caffe.cpp:275] Batch 15, loss = 0.0362216
I0816 14:40:29.873572 2801 caffe.cpp:275] Batch 16, accuracy = 0.98
I0816 14:40:29.873939 2801 caffe.cpp:275] Batch 16, loss = 0.0458992
I0816 14:40:29.896610 2801 caffe.cpp:275] Batch 17, accuracy = 0.98
I0816 14:40:29.896783 2801 caffe.cpp:275] Batch 17, loss = 0.0315805
I0816 14:40:29.918918 2801 caffe.cpp:275] Batch 18, accuracy = 1
I0816 14:40:29.919095 2801 caffe.cpp:275] Batch 18, loss = 0.00636367
I0816 14:40:29.938529 2801 caffe.cpp:275] Batch 19, accuracy = 0.98

```

```
I0816 14:40:29.941370 2801 caffe.cpp:275] Batch 19, loss = 0.0730766
I0816 14:40:29.963331 2801 caffe.cpp:275] Batch 20, accuracy = 0.98
I0816 14:40:29.963515 2801 caffe.cpp:275] Batch 20, loss = 0.0758103
I0816 14:40:29.986037 2801 caffe.cpp:275] Batch 21, accuracy = 0.96
I0816 14:40:29.986402 2801 caffe.cpp:275] Batch 21, loss = 0.0655355
I0816 14:40:30.008844 2801 caffe.cpp:275] Batch 22, accuracy = 0.98
I0816 14:40:30.009043 2801 caffe.cpp:275] Batch 22, loss = 0.0565184
I0816 14:40:30.030681 2801 caffe.cpp:275] Batch 23, accuracy = 0.98
I0816 14:40:30.030829 2801 caffe.cpp:275] Batch 23, loss = 0.0290294
I0816 14:40:30.052981 2801 caffe.cpp:275] Batch 24, accuracy = 0.97
I0816 14:40:30.053589 2801 caffe.cpp:275] Batch 24, loss = 0.0396222
I0816 14:40:30.075492 2801 caffe.cpp:275] Batch 25, accuracy = 0.99
I0816 14:40:30.075613 2801 caffe.cpp:275] Batch 25, loss = 0.0805439
I0816 14:40:30.099077 2801 caffe.cpp:275] Batch 26, accuracy = 0.99
I0816 14:40:30.099236 2801 caffe.cpp:275] Batch 26, loss = 0.114673
I0816 14:40:30.121295 2801 caffe.cpp:275] Batch 27, accuracy = 1
I0816 14:40:30.121448 2801 caffe.cpp:275] Batch 27, loss = 0.01823
I0816 14:40:30.143508 2801 caffe.cpp:275] Batch 28, accuracy = 0.98
I0816 14:40:30.143793 2801 caffe.cpp:275] Batch 28, loss = 0.0537291
I0816 14:40:30.165635 2801 caffe.cpp:275] Batch 29, accuracy = 0.97
I0816 14:40:30.166054 2801 caffe.cpp:275] Batch 29, loss = 0.110882
I0816 14:40:30.188333 2801 caffe.cpp:275] Batch 30, accuracy = 1
I0816 14:40:30.188475 2801 caffe.cpp:275] Batch 30, loss = 0.0232105
I0816 14:40:30.211480 2801 caffe.cpp:275] Batch 31, accuracy = 1
I0816 14:40:30.211638 2801 caffe.cpp:275] Batch 31, loss = 0.0020194
I0816 14:40:30.233865 2801 caffe.cpp:275] Batch 32, accuracy = 1
I0816 14:40:30.234025 2801 caffe.cpp:275] Batch 32, loss = 0.0107053
I0816 14:40:30.255594 2801 caffe.cpp:275] Batch 33, accuracy = 1
I0816 14:40:30.256156 2801 caffe.cpp:275] Batch 33, loss = 0.00484999
I0816 14:40:30.277842 2801 caffe.cpp:275] Batch 34, accuracy = 0.99
I0816 14:40:30.278465 2801 caffe.cpp:275] Batch 34, loss = 0.0479895
I0816 14:40:30.298285 2801 caffe.cpp:275] Batch 35, accuracy = 0.95
I0816 14:40:30.301501 2801 caffe.cpp:275] Batch 35, loss = 0.165702
I0816 14:40:30.322823 2801 caffe.cpp:275] Batch 36, accuracy = 1
I0816 14:40:30.323734 2801 caffe.cpp:275] Batch 36, loss = 0.0041475
I0816 14:40:30.345669 2801 caffe.cpp:275] Batch 37, accuracy = 0.97
I0816 14:40:30.346505 2801 caffe.cpp:275] Batch 37, loss = 0.0726677
I0816 14:40:30.367012 2801 caffe.cpp:275] Batch 38, accuracy = 0.99
I0816 14:40:30.368809 2801 caffe.cpp:275] Batch 38, loss = 0.0187083
I0816 14:40:30.391441 2801 caffe.cpp:275] Batch 39, accuracy = 0.98
I0816 14:40:30.391620 2801 caffe.cpp:275] Batch 39, loss = 0.0569579
I0816 14:40:30.413909 2801 caffe.cpp:275] Batch 40, accuracy = 0.99
I0816 14:40:30.414444 2801 caffe.cpp:275] Batch 40, loss = 0.0390845
I0816 14:40:30.436063 2801 caffe.cpp:275] Batch 41, accuracy = 0.98
I0816 14:40:30.436671 2801 caffe.cpp:275] Batch 41, loss = 0.0840227
I0816 14:40:30.458598 2801 caffe.cpp:275] Batch 42, accuracy = 0.98
I0816 14:40:30.458744 2801 caffe.cpp:275] Batch 42, loss = 0.0354611
I0816 14:40:30.481271 2801 caffe.cpp:275] Batch 43, accuracy = 1
I0816 14:40:30.481432 2801 caffe.cpp:275] Batch 43, loss = 0.00441331
I0816 14:40:30.504631 2801 caffe.cpp:275] Batch 44, accuracy = 0.99
I0816 14:40:30.504773 2801 caffe.cpp:275] Batch 44, loss = 0.0300453
```

```

I0816 14:40:30.526890 2801 caffe.cpp:275] Batch 45, accuracy = 0.99
I0816 14:40:30.527060 2801 caffe.cpp:275] Batch 45, loss = 0.0339
I0816 14:40:30.548745 2801 caffe.cpp:275] Batch 46, accuracy = 1
I0816 14:40:30.549238 2801 caffe.cpp:275] Batch 46, loss = 0.00635041
I0816 14:40:30.571394 2801 caffe.cpp:275] Batch 47, accuracy = 0.99
I0816 14:40:30.571542 2801 caffe.cpp:275] Batch 47, loss = 0.0192659
I0816 14:40:30.594262 2801 caffe.cpp:275] Batch 48, accuracy = 0.96
I0816 14:40:30.594359 2801 caffe.cpp:275] Batch 48, loss = 0.0596552
I0816 14:40:30.616652 2801 caffe.cpp:275] Batch 49, accuracy = 1
I0816 14:40:30.616801 2801 caffe.cpp:275] Batch 49, loss = 0.00606366 //
共 50 个 batch
I0816 14:40:30.616874 2801 caffe.cpp:280] Loss: 0.0432061
I0816 14:40:30.616946 2801 caffe.cpp:292] accuracy = 0.9856
I0816 14:40:30.617023 2801 caffe.cpp:292] loss = 0.0432061 (* 1 = 0.0432061 loss)


```

从上面的打印输出可以看出，测试数据中的 accuracy 平均成功率为 98%。

5.3.2 mnist 手写测试

上一节的内容使用 caffe 自带的测试数据验证训练出来的 LeNet 网络，本节我们用手写的数字输入到网络进行分类识别，手写数字的图片必须满足以下条件：

- (1) 必须是 256 位黑白色；
- (2) 必须是黑底白字；
- (3) 像素大小必须是 28×28 ；
- (4) 数字在图片中间，上下左右没有过多的空白。

例如测试数字“3”的图片  为例，图片文件名“again.bmp”，存放目录是 caffe-master/examples/images/again.bmp，如下是用 Python 编写手写数字识别脚本和注释。

```

import os
import sys
import numpy as np          // import Numpy 数值计算库
import matplotlib.pyplot as plt
caffe_root = '/home/alex/caffe-master/' // 设置 caffe_root 目录，请读者根据环
境设置
sys.path.insert(0, caffe_root + 'python') // 添加系统环境变量
import caffe                // import caffe 模块
MODEL_FILE = '/home/alex/caffe-master/examples/mnist/lenet.prototxt' //

```


指定 LeNet 网络定义文件

```
PRETRAINED = '/home/alex/caffe-master/examples/mnist/lenet_iter_10000.
caffemodel' //训练好的 model 文件
```

```
IMAGE_FILE = '/home/alex/caffe-master/examples/images/again.bmp' //测试图
片路径
```

```
input_image=caffe.io.load_image(IMAGE_FILE,color=False) // caffe 接口载入
文件
```

```
#print input_image
net = caffe.Classifier(MODEL_FILE, PRETRAINED) // 载入 LeNet 分类器
```

```
prediction=net.predict([input_image], oversample = False) // 预测图片, 进
行分类; 没有 crop 时, oversample 过采样为 false
```

```
caffe.set_mode_cpu() // 设置为 CPU 模式
```

```
print 'predicted class:', prediction[0].argmax() //打印出分类结果
```

在 Caffe 根目录下执行上述脚本, 输出的内容和注释如下:

```
I0525 16:14:28.662001 17373 layer_factory.hpp:77] Creating layer data //
创建 data 层
I0525 16:14:28.662029 17373 net.cpp:91] Creating Layer data
I0525 16:14:28.662046 17373 net.cpp:399] data -> data
I0525 16:14:28.662067 17373 net.cpp:141] Setting up data // 准备输入数据
I0525 16:14:28.662068 17373 net.cpp:148] Top shape: 64 1 28 28 (50176)
//batchsize = 64, 通道 1, 28*28 的输入, 共 50176 输入, 表 5-1 所示。
I0525 16:14:28.662068 17373 net.cpp:156] Memory required for data: 200704
I0525 16:14:28.662068 17373 layer_factory.hpp:77] Creating layer conv1 //
创建卷积层
I0525 16:14:28.662068 17373 net.cpp:91] Creating Layer conv1
I0525 16:14:28.662070 17373 net.cpp:425] conv1 <- data
I0525 16:14:28.662070 17373 net.cpp:399] conv1 -> conv1
I0525 16:14:28.662070 17373 net.cpp:141] Setting up conv1
I0525 16:14:28.662156 17373 net.cpp:148] Top shape: 64 20 24 24 (737280)
//batchsize=64, 通道 20, 24*24 输入, 共 737280 个输入, 表 5-1 所示。
I0525 16:14:28.662238 17373 net.cpp:156] Memory required for data: 3149824
I0525 16:14:28.662307 17373 layer_factory.hpp:77] Creating layer pool1
I0525 16:14:28.662374 17373 net.cpp:91] Creating Layer pool1
I0525 16:14:28.662394 17373 net.cpp:425] pool1 <- conv1
I0525 16:14:28.662411 17373 net.cpp:399] pool1 -> pool1
I0525 16:14:28.662433 17373 net.cpp:141] Setting up pool1
I0525 16:14:28.662451 17373 net.cpp:148] Top shape: 64 20 12 12 (184320)
I0525 16:14:28.662467 17373 net.cpp:156] Memory required for data: 3887104
I0525 16:14:28.662482 17373 layer_factory.hpp:77] Creating layer conv2
I0525 16:14:28.662499 17373 net.cpp:91] Creating Layer conv2
I0525 16:14:28.662514 17373 net.cpp:425] conv2 <- pool1
I0525 16:14:28.662531 17373 net.cpp:399] conv2 -> conv2
I0525 16:14:28.662678 17373 net.cpp:141] Setting up conv2
I0525 16:14:28.662698 17373 net.cpp:148] Top shape: 64 50 8 8 (204800)
```



```

I0525 16:14:28.662714 17373 net.cpp:156] Memory required for data: 4706304
I0525 16:14:28.662731 17373 layer_factory.hpp:77] Creating layer pool2
I0525 16:14:28.662750 17373 net.cpp:91] Creating Layer pool2
I0525 16:14:28.662765 17373 net.cpp:425] pool2 <- conv2
I0525 16:14:28.662781 17373 net.cpp:399] pool2 -> pool2
I0525 16:14:28.662798 17373 net.cpp:141] Setting up pool2
I0525 16:14:28.662816 17373 net.cpp:148] Top shape: 64 50 4 4 (51200)
I0525 16:14:28.662830 17373 net.cpp:156] Memory required for data: 4911104
I0525 16:14:28.662844 17373 layer_factory.hpp:77] Creating layer ip1
I0525 16:14:28.662863 17373 net.cpp:91] Creating Layer ip1
I0525 16:14:28.662881 17373 net.cpp:425] ip1 <- pool2
I0525 16:14:28.662897 17373 net.cpp:399] ip1 -> ip1
I0525 16:14:28.664897 17373 net.cpp:141] Setting up ip1
I0525 16:14:28.664926 17373 net.cpp:148] Top shape: 64 500 (32000)
I0525 16:14:28.664940 17373 net.cpp:156] Memory required for data: 5039104
I0525 16:14:28.664958 17373 layer_factory.hpp:77] Creating layer relul
I0525 16:14:28.664976 17373 net.cpp:91] Creating Layer relul
I0525 16:14:28.664991 17373 net.cpp:425] relul <- ip1
I0525 16:14:28.665007 17373 net.cpp:386] relul -> ip1 (in-place)
I0525 16:14:28.665025 17373 net.cpp:141] Setting up relul
I0525 16:14:28.665041 17373 net.cpp:148] Top shape: 64 500 (32000)
I0525 16:14:28.665062 17373 net.cpp:156] Memory required for data: 5167104
I0525 16:14:28.665078 17373 layer_factory.hpp:77] Creating layer ip2
I0525 16:14:28.665096 17373 net.cpp:91] Creating Layer ip2
I0525 16:14:28.665110 17373 net.cpp:425] ip2 <- ip1
I0525 16:14:28.665127 17373 net.cpp:399] ip2 -> ip2
I0525 16:14:28.665174 17373 net.cpp:141] Setting up ip2
I0525 16:14:28.665194 17373 net.cpp:148] Top shape: 64 10 (640)
I0525 16:14:28.665208 17373 net.cpp:156] Memory required for data: 5169664
I0525 16:14:28.665225 17373 layer_factory.hpp:77] Creating layer prob
I0525 16:14:28.665241 17373 net.cpp:91] Creating Layer prob
I0525 16:14:28.665256 17373 net.cpp:425] prob <- ip2
I0525 16:14:28.665272 17373 net.cpp:399] prob -> prob
I0525 16:14:28.665292 17373 net.cpp:141] Setting up prob
I0525 16:14:28.665307 17373 net.cpp:148] Top shape: 64 10 (640)
I0525 16:14:28.665323 17373 net.cpp:156] Memory required for data: 5172224
I0525 16:14:28.665336 17373 net.cpp:219] prob does not need backward computation.
I0525 16:14:28.665351 17373 net.cpp:219] ip2 does not need backward computation.
I0525 16:14:28.665365 17373 net.cpp:219] relul does not need backward computation.
I0525 16:14:28.665380 17373 net.cpp:219] ip1 does not need backward computation.
I0525 16:14:28.665395 17373 net.cpp:219] pool2 does not need backward computation.
I0525 16:14:28.665408 17373 net.cpp:219] conv2 does not need backward computation.
I0525 16:14:28.665423 17373 net.cpp:219] pool1 does not need backward computation.
I0525 16:14:28.665539 17373 net.cpp:219] conv1 does not need backward computation.
I0525 16:14:28.665604 17373 net.cpp:219] data does not need backward computation.
I0525 16:14:28.665665 17373 net.cpp:261] This network produces output prob
I0525 16:14:28.665732 17373 net.cpp:274] Network initialization done.
I0525 16:14:28.668491 17373 net.cpp:753] Ignoring source layer mnist
I0525 16:14:28.668762 17373 net.cpp:753] Ignoring source layer loss
predicted class: 3

```

最后识别出手写数字是“3”，成功识别图片中的数字。

5.3.3 mnist 样本字库的图片转换

有时为了方便研究和分析样本训练数据或测试样本数据，需要将测试样本库文件转换成图片，下面的 Python 脚本完成了测试样本库格式转换成 bmp 格式图片，训练样本库的转换方法只需要将“filename”变量定义为“train-images-idx3-ubyte.gz”即可。

```
import numpy as np
import struct
import matplotlib.pyplot as plt
import Image // 载入图片库

filename = 't10k-images-idx3-ubyte' //压缩格式的测试样本库
binfile = open(filename, 'rb') // 以bin 格式打开文件
buf = binfile.read() // 读全部文件内容到缓存 buf

index = 0
magic, numImages, numRows, numColumns = struct.unpack_from('>IIII', buf,
index) //指定类型读数据，得到图片总数
index += struct.calcsize('>IIII')

for image in range(0, numImages): // 循环读图片
    im = struct.unpack_from('>784B', buf, index) //读一个图片
    index += struct.calcsize('>784B')

    im = np.array(im, dtype='uint8')
    im = im.reshape(28, 28) //28*28 大小像素

    #fig = plt.figure()
    #plotwindow = fig.add_subplot(111)
    #plt.imshow(im, cmap='gray')
    #plt.show()
    im = Image.fromarray(im)
    im.save('mnist_test/train_%s.bmp'%image, 'bmp') //保存转换后的图片文件
```

参考文献

- [1] Gradient-Based Learning Applied to Document Recognition Yan LeCun <http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf> 1989

6

第 6 章 AlexNet 模型

上一章我们介绍了 LeNet 经典网络模型及构成要素和设计方法，此模型对手写数字识别具有较高的识别率，但对大数据量、复杂的物体图片分类还远远不够。在 2012 ILSVRC 竞赛（Large Scale Visual Recognition Challenge）中，AlexNet 模型赢得了第一名，Top-5 错误率 15.3%，比上一年冠军下降了十个百分点。从此，AlexNet 成为 CNN 领域内具有重要历史意义的一个网络模型。AlexNet 模型证明了 CNN 在复杂模型下的有效性，并使用 GPU 使大数据训练在可接受的时间范围内得到了结果。

本章先介绍 AlexNet 网络模型并进行解读，然后分析了 AlexNet 模型之所以能够取得重大成功的原因。最后详细介绍此模型在 ILSVRC2012 数据集下 Caffe 的训练方法。

6.1 AlexNet 模型介绍

2012 年，Geoffrey 和他的学生 Alex 为了回应质疑，在 ILSVRC 竞赛中出手，刷新了 Image Classification 的纪录，一举奠定了 Deep Learning 在计算机视觉中的地位。在这次竞赛中 Alex 所用的网络结构被称为 AlexNet 模型。

根据 Alex 2012 年 NIPS（Conference and Workshop on Neural Information Processing

Systems, 神经信息处理系统大会) 公开发表的论文 “ImageNet classification with deep convolutional neural networks” 的内容^[1], AlexNet 网络的基本结构如图 6-1 所示。

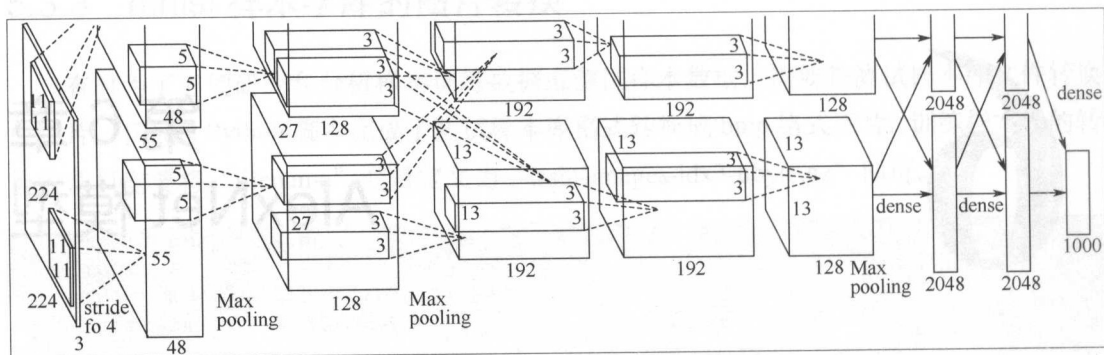


图 6-1 AlexNet 网络基本结构

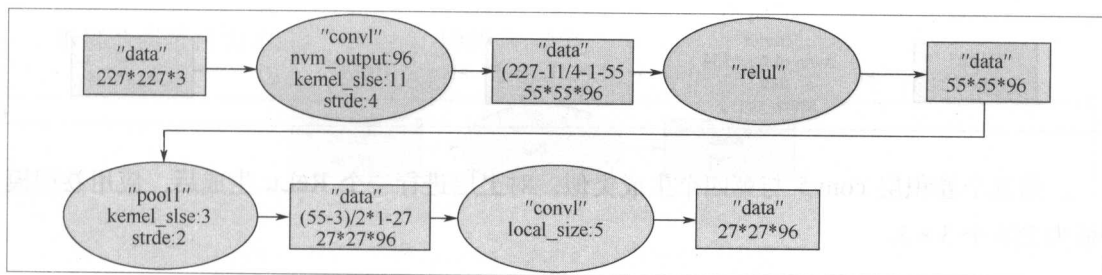
6.2 AlexNet 模型解读

AlexNet 共有八层, 有 60M 以上的参数量。其前五层是卷积层, 后三层是全连接层, 最后一个全连接层的输出具有 1000 个输出的 softmax。网络最后的优化目标是最大化平均的 multinomial logistic regression。

第一个卷积层 conv1 中, AlexNet 采用了 96 个 $11 \times 11 \times 3$ 的 kernel。在 stride 为 4 的情况下对于 $224 \times 224 \times 3$ 的图像进行了滤波。换句话说, 就是采用了 11×11 的卷积模板在三个通道上, 间隔为 4 个像素的采样频率上对于图像进行了卷积操作。

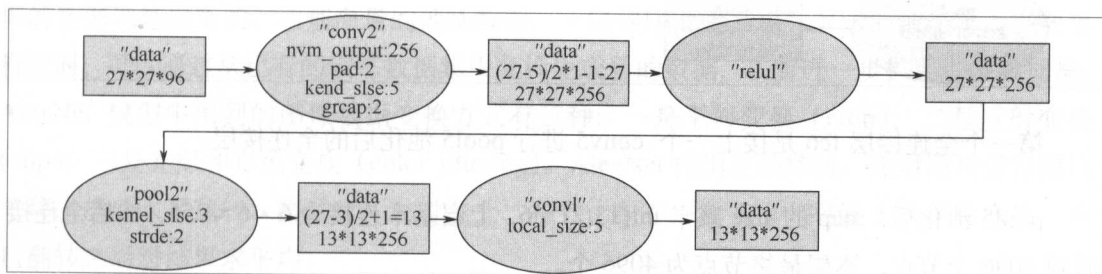
最初的输入神经元的个数为 $224 \times 224 \times 3 = 150528$ 个。对于每一个 map 来说, 间隔为 4, 因此 $224/4 = 56$, 然后减去边缘的一个为 55 个, 也就是本层的 map 大小为 55×55 。因此, 神经元数目为 $55 \times 55 \times 96 = 290400$ 。

得到基本的卷积数据后, 先进行一次 ReLU (relu1) 以及 Norm (norm1) 变换, 然后进行 pooling (pool1), 作为输出传递到下一层。本层 map 数目为 96 个。



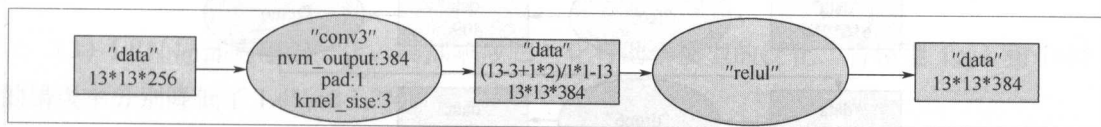
第二个卷积层 conv2 是 conv1 经过 norm 以及 pool 池化后, 然后再应用 256 个 5×5 的卷积模板卷积后的结果。

pool1 池化后, map 的大小减半 $\text{int}(55/2) = 27$, 故而得到的本层的神经元数目为 $27 \times 27 \times 256 = 186642$ 个。本层 map 数目为 256 个。



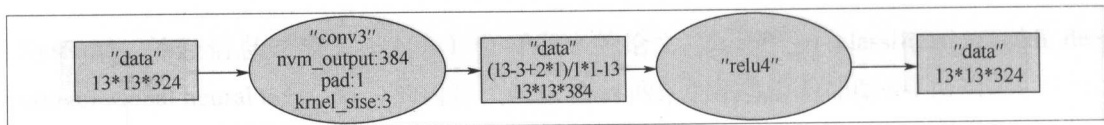
第三个卷积层 conv3 的生成过程和第二层类似, 不同的是这一层应用的是 384 个 3×3 的卷积模板得到的。

pool2 池化后, map 的大小减半 $\text{int}(27/2) = 13$, 故而得到的本层的神经元数目为 $13 \times 13 \times 384 = 64896$ 个。本层 map 数目为 384 个。



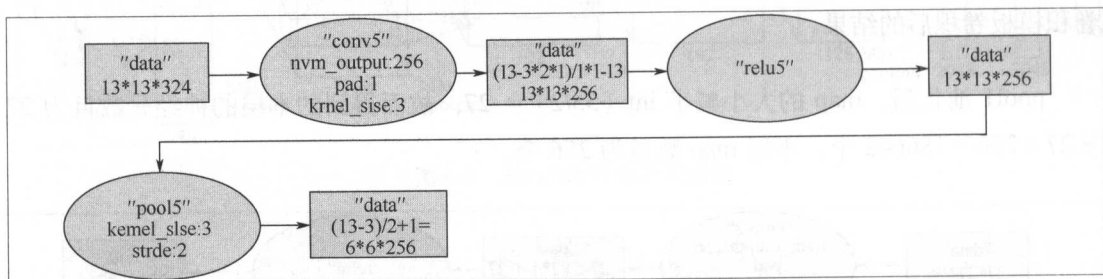
第四个卷积层 conv4 是 conv3 进行一次 ReLU 后, 应用的是 384 个 3×3 的卷积模板得到的。

本层的神经元数目为 $13 \times 13 \times 384 = 64896$ 个。本层 map 数目为 384 个, size 为 13×13 。



第五个卷积层 conv5 与第四个生成类似，对上层进行一个 ReLu 生成后，应用卷积模板为 256 个 3×3 。

本层神经元的数目为 $13 \times 13 \times 256 = 43264$ ，map 数目为 256 个，size 是 13×13 。

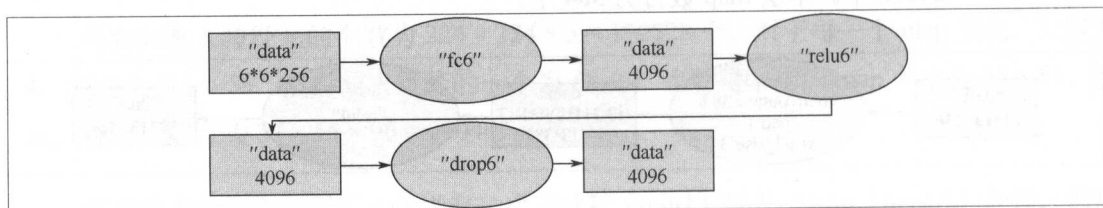


第一个全连接层 fc6 是接上一个 conv5 进行 pool5 池化后的全连接层。

pool5 池化后，map 的 size 减半 $\text{int}(13/2) = 6$ ，上层基本连接为 $6 \times 6 \times 256$ ，然后全连接后到 4096 个节点，本层最终节点为 4096 个。

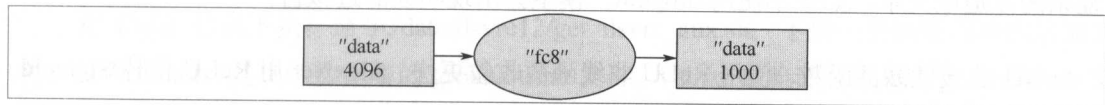
第二个全连接层 fc7 是上一个 fc6 进行 ReLU(relu6)后，然后进行 dropout(drop6)后再进行全连接的结果。

本层节点数目为 4096 个。



最后一个全连接层 fc8 则是上一个 fc7 再次进行 ReLU(relu7)以及 dropout(drop7)后再进行全连接的结果。最后输出为融合 label 的 softmax loss。

本层节点数目为 1000 个，对应 1000 类对象。



6.3 AlexNet 模型特点

AlexNet 作为 2012 年 ILSVRC 竞赛的冠军，能够取得成功，这是因为 AlexNet 有以下创新点：

(1) 大数据训练，百万级 ImageNet 图像数据输入。一般的观点认为神经网络是靠足够多的数据训练出来的。增加海量的训练数据，一定程度上能够提升算法的准确率。当数据有限时，可以通过从已有的训练数据集中变换生成新的数据，从而进一步扩大训练数据量。AlexNet 模型中用到的图像数据变换方式有三种：一是平移变换 (crop)；二是反射变换 (flip)；三是光照和彩色变换 (color jittering)。AlexNet 模型在训练时，先对图片进行随机平移，然后水平翻转。测试时，先对左上、右上、左下、右下和中间做 5 次平移变换，然后翻转之后对结果求平均。

(2) 多 GPU 训练。由于单 GPU 的存储空间过小，如 GTX580 GPU 只有 3GB 内存，这限制了其训练网络的最大规模。ImageNet 数据集中 120 万张图片训练的网络规模超出了单个 GPU 的存储能力。因此使用两块 GPU，在每块 GPU 上存储一半的 kernels，这两块 GPU 只在特定的层上通信。比起在一个 GPU 上训练的每个卷积层只有一半 kernels 的方案，Top-1 和 Top-5 的错误率分别降低了 1.7% 和 1.2%。

(3) LRN 局部响应归一化。局部响应归一化有助于模型的泛化，将模型 Top1 和 Top5 的错误率分别降低了 1.4% 和 1.2%。

(4) 重叠池化。将模型 Top1 和 Top5 的错误率分别降低了 0.4% 和 0.3%。

(5) 避免过拟合。AlexNet 提出了一个非常有效的模型组合，这种技术是 Dropout。以这种方式 “dropout” 的神经元既不参与前向传播，也不参与反向传播。所以每次输入一个

样本，相当于神经网络尝试了一个新的结构，这些结构之间共享权重，从而降低了神经元复杂的互适应关系。如果不使用 dropout，模型会出现明显的过拟合。

(6) 非线性激活函数，使用 ReLU 非线性函数收敛更快。AlexNet 用 ReLU 代替 Sigmoid，是因为 ReLU 得到 SGD 的收敛速度比 sigmoid/tanh 快很多。主要原因是 ReLU 是线性的、非饱和的，而且 ReLU 只需要一个阈值就可以得到激活值。

6.4 Caffe 环境 AlexNet 模型训练

6.4.1 数据准备

首先到 ImageNet 官方网站下载 ILSVRC2012 的训练数据集和验证数据集，地址是：<http://www.image-net.org/signup.php?next=download-images>。下载前需要邮箱注册，邮箱地址不能以.com 为后缀。

下载后有两个压缩文件：ILSVRC2012_img_train.tar 是训练数据集，ILSVRC2012_img_val.tar 是测试数据集，将这两个文件拷贝到 examples/imagenet 目录下，并在当前目录下解压，解压命令如下：

```
$tar -xvf ILSVRC2012_img_train.tar ./train
$tar -xvf ILSVRC2012_img_val.tar ./val
```

ILSVRC2012_img_train.tar 解压后是 1000 个 tar 文件，每个 tar 文件表示 1000 个分类中的一个类。需要对这 1000 个 tar 文件再次解压。在 train 目录下执行 unzip.sh 文件，最后得到 1000 个文件夹。每个文件夹中是该类的图片。ILSVRC2012_img_val.tar 解压后的文件夹包含了所有的测试集图片。unzip.sh 脚本内容如下：

```
dir=./
for x in `ls *.tar`
do
filename=`basename $x .tar`
mkdir $filename
tar -xvf $x -C ./ $filename
```

6.4.2 其他支持文件

在 Caffe 目录下执行命令 `./data/ilsrvrc12/get_ilsrvrc_aux.sh`, 下载一些训练需要的附加文件。其中 `train.txt` 是训练数据集的 ground truth 文件, `val.txt` 是验证数据集的 ground truth 文件。这两个文件的作用是在生成 LMDB 数据库时提供标签信息。

6.4.3 图片预处理

1. 大小归一化

将所有的图片都归一化为 256×256 的大小, 对于一个长方形图片, 首先将短边变成 256 的长度, 然后剪裁图片中心的 256×256 部分。

接下来, 在 `examples/imagenet/create_imagenet.sh` 文件中, 将 `RESIZE=false` 更改为 `RESIZE= true`, 将所有图片归一化为 25×256 的大小。注意需将文件中的训练数据集和测试数据集的地址更改为服务器中实际存放的地址, 即文件中设置 “`TRAIN_DATA_ROOT=/examples/ imagenet/train/`”, `VAL_DATA_ROOT=/dataset/imagenet/val/`”。执行该文件后生成训练数据和测试数据的 LMDB 数据库: `ilsrvrc12_train_lmdb`、`ilsrvrc12_val_lmdb`。

2. 减去像素平均值

所有图片的每个像素都减去所有训练集图片的平均值。训练集图片的平均值存储于 `data/ilsrvrc12/imagenet_mean.binaryproto`。如果没有该文件, 执行 `./examples/imagenet/make_imagenet_mean.sh` 脚本可以生成该文件。

6.4.4 ImageNet 数据集介绍

ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 是 Pascal Visual Object Challenge 的子部分。ImageNet 是一个数据库, 有超过 22,000 个种类, 超过 1,500 万张图片。

ILSVRC 使用 1,000 个类, 每个类中有 1,000 个图片。这 1,000 个类——对应于 WorldNet 的 1000 个同义子集。由于这 1,000 个同义子集互相不重叠, 也不是祖先, 可称这样的子集

为低层子集，它们构成了 ImageNet 层次结构的一部分。这个子集的所有祖先共有 860 个，被称为高层子集。在层次结构中，所有的低层特征称为叶节点，高层子集称为中间节点。尽管在 ImageNet 中低层子集会有子类，但 ILSVRC2012 并不考虑这些子类，ILSVRC2012 的层次结构可以看作是对完整 ImageNet 结构的剪裁。ILSVRC 竞赛中，所有的标记都是针对低层子集的，参赛者必须预测这 1,000 个低层子集标记之一，不考虑预测高层子集的结果，而且也没有高层子集的训练图片。

在 ILSVRC 中，子集信息可参见 ILSVRC2012_devkit_t12 中的 data/meta.mat 文件中的矩阵。矩阵中的每一个行对应于一个子集，每一项中包含如下域值：

- ILSVRC2012_ID 是为每一子集分配的一个整数 ID 值，所有低层子集的 ID 值都在 1 到 1000 之间，所有高层子集的 ID 值都大于 1000。所有的子集都对其 ID 值进行排序。提交预测结果时，ILSVRC2012_ID 也作为相应子集的标记。
- WNID 是子集在 WordNet 中的 ID。用于在 ImageNet 或 WorldNet 中唯一标记一个子集。包含训练图片的 tar 文件就是用 WNID 命名的。同样每一个训练图片也是用 WNID 命名的。
- num_children 是子集在剪裁后的结构中子孙的数目。对于低层子集其值为 0，对于高层子集的值不为 0。
- children 是子孙子集的 ILSVRC2012_ID 的向量。
- wordnet_height 是完整的 ImageNet/WorldNet 层次结构中到叶节点的最长路径的值。（完整的 ImageNet/WorldNet 层次结构中叶节点的 wordnet_height 值为 0。）

Caffe 使用的标签和 ILSVRC2012_devkit 是不一致的。ILSVRC2012_ID 是 ILSVRC2012_devkit 的提供的编号。而 Caffe 中图片的标签是以图片所属子集的名字的 ASCII 的顺序排列的，并依次从 0 到 999 编号。本章所提供的所有示例都是依据 Caffe 的编号编写的。在 Caffe 目录的 data/ilsvrc12/synset_words.txt 文件内，可以查看子集/编号的对应关系。

6.4.5 ImageNet 图片介绍

对于每一个子集都有一个 tar 文件, 用 WNID 命名。而图片文件, 则命名为 x_y .JPEG。其中 x 是子集的 WNID, y 是整数 (不是固定长度, 而且不一定连续), 所有的图片都是 JPEG 格式, 共有 1,281,167 张训练图片。其中每个子集的训练图片数目在 732 至 1,300 之间。验证图片共有 50,000 张验证图片, 分别被命名为:

```
ILSVRC2012_val_00000001.JPEG
ILSVRC2012_val_00000002.JPEG
...
ILSVRC2012_val_00049999.JPEG
ILSVRC2012_val_00050000.JPEG
```

对于每一个子集分别有 50 张验证图片。验证图片的 ground truth 在 data/ILSVRC2012_validation_ground_truth.txt 中, 文件中每一行包含一个图片对应的 ILSVRC2012_ID, 并以图片名称的升序排列。测试图片共有 100,000 张测试图片, 测试图片的命名如下:

```
ILSVRC2012_test_00000001.JPEG
ILSVRC2012_test_00000002.JPEG
...
ILSVRC2012_test_00099999.JPEG
ILSVRC2012_test_00100000.JPEG
```

对于每一个子集分别有 100 张测试图片。

6.4.6 ImageNet 模型训练

AlexNet 模型定义在文件 models/bvlc_alexnet/train_val.prototxt 中, 我们首先需要将文件中的训练数据集和测试数据集的地址更改为服务器中实际存放的地址。训练参数定义于文件 models/bvlc_alexnet/solver.prototxt 中。

在 Caffe 目录下执行命令:

```
./build/tools/caffe train --solver=models/bvlc_alexnet/solver.prototxt
```

由于训练数据量大, 时间相对比较长, 可以通过后台运行, 命令为:

```
$nohup ./build/tools/caffe train --solver=models/bvlc_alexnet/solver.prototxt &
```

该命令的输出会定向到 nohup.out 文件中, 可以查看此文件模型训练的情况。查看 GPU 是否被占用以及被占用的内存大小命令:

```
$nvidia-smi
```

训练部分输出如下:

```
# ./build/tools/caffe train --solver=models/bvlc_alexnet/solver.prototxt
I0819 15:18:11.564997 4699 caffe.cpp:178] Use GPU.
I0819 15:18:11.565414 4699 solver.cpp:48] Initializing solver from parameters:
//初始化网络参数
test_iter: 1000
test_interval: 1000
base_lr: 0.01 // 开始的学习率
display: 20 // 每 20 次打印显示 loss
max_iter: 450000 // train 最大迭代 450000
lr_policy: "step" // 学习率的 drop 是以 gamma 在每一次迭代中
gamma: 0.1
momentum: 0.9
weight_decay: 0.0005
stepsize: 100000 // 每个步长的迭代降低学习率*gamma
snapshot: 10000 // 每迭代 10000 次, 保存一次快照
snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU // 使用 GPU 模式
net: "models/bvlc_alexnet/train_val.prototxt"
I0819 15:18:11.565773 4699 solver.cpp:91] Creating training net from net
file: models/bvlc_alexnet/train_val.prototxt
I0819 15:18:11.566068 4699 net.cpp:313] The NetState phase (0) differed from
the phase (1) specified by a rule in layer data
I0819 15:18:11.566102 4699 net.cpp:313] The NetState phase (0) differed from
the phase (1) specified by a rule in layer accuracy
I0819 15:18:11.566201 4699 net.cpp:49] Initializing net from parameters:
name: "AlexNet"
state {
  phase: TRAIN // 表明训练阶段执行
}
layer { // 数据层
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param { // 对数据进行预处理
    mirror: true // 是否做镜像
    crop_size: 227 // 平移变换大小
```



```

    mean_file: "data/ilsrvrc12/imagenet_mean.binaryproto" // 均值文件
  }
  data_param {          // 设定数据来源
    source: "examples/imagenet/ilsrvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1          // 学习率
    decay_mult: 1       // 权值衰减
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96      // 卷积核 (filter) 个数 96
    kernel_size: 11     // 卷积核的大小 11
    stride: 4           // 卷积核的步长 4
    weight_filler {
      type: "gaussian"  // 权重初始化类型为与 gaussian
      std: 0.01
    }
    bias_filler {      // 偏置项初始化, 一般为 constant, 值全为 0
      type: "constant"
      value: 0
    }
  }
}
}
layer {                // ReLU 层
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {                // LRN 层
  name: "norm1"
  type: "LRN"
  bottom: "conv1"
  top: "norm1"
  lrn_param {          // 归一化公式的参数
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}

```

```

    }
}
layer {                                // Pooling 层
    name: "pool1"
    type: "Pooling"
    bottom: "norm1"
    top: "pool1"
    pooling_param {
        pool: MAX                      // 池化方法 MAX
        kernel_size: 3                // 核大小 3
        stride: 2                      // 步长为 2
    }
}
layer {                                // 卷积层, 参数含义同上
    name: "conv2"
    type: "Convolution"
    bottom: "pool1"
    top: "conv2"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256
        pad: 2
        kernel_size: 5
        group: 2
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler {
            type: "constant"
            value: 0.1
        }
    }
}
layer {
    name: "relu2"
    type: "ReLU"
    bottom: "conv2"
    top: "conv2"
}
layer {                                // LRN 层, 参数含义同上
    name: "norm2"
    type: "LRN"
    bottom: "conv2"

```



```

    top: "norm2"
    lrn_param {
      local_size: 5
      alpha: 0.0001
      beta: 0.75
    }
  }
  layer {                                // Pooling 层, 参数含义同上
    name: "pool2"
    type: "Pooling"
    bottom: "norm2"
    top: "pool2"
    pooling_param {
      pool: MAX
      kernel_size: 3
      stride: 2
    }
  }
  layer {                                // 卷积层, 参数含义同上
    name: "conv3"
    type: "Convolution"
    bottom: "pool2"
    top: "conv3"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    convolution_param {
      num_output: 384
      pad: 1
      kernel_size: 3
      weight_filler {
        type: "gaussian"
        std: 0.01
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
  layer {
    name: "relu3"
    type: "ReLU"
    bottom: "conv3"
    top: "conv3"
  }
}

```

```

layer {                                // 卷积层, 参数含义同上
    name: "conv4"
    type: "Convolution"
    bottom: "conv3"
    top: "conv4"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 384
        pad: 1
        kernel_size: 3
        group: 2
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler {
            type: "constant"
            value: 0.1
        }
    }
}
layer {
    name: "relu4"
    type: "ReLU"
    bottom: "conv4"
    top: "conv4"
}
layer {                                // 卷积层, 参数含义同上
    name: "conv5"
    type: "Convolution"
    bottom: "conv4"
    top: "conv5"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256
        pad: 1
        kernel_size: 3

```

```

    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
layer {
  name: "relu5"
  type: "ReLU"
  bottom: "conv5"
  top: "conv5"
}
layer {                                     // Pooling 层, 参数含义同上
  name: "pool5"
  type: "Pooling"
  bottom: "conv5"
  top: "pool5"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {                                     // 全连接层
  name: "fc6"
  type: "InnerProduct"
  bottom: "pool5"
  top: "fc6"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 4096
    weight_filler {
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}

```



```

    }
  }
  layer {
    name: "relu6"
    type: "ReLU"
    bottom: "fc6"
    top: "fc6"
  }
  layer { // Dropout 层
    name: "drop6"
    type: "Dropout"
    bottom: "fc6"
    top: "fc6"
    dropout_param {
      dropout_ratio: 0.5 // 丢弃数据的概率
    }
  }
  layer { // 全连接层
    name: "fc7"
    type: "InnerProduct"
    bottom: "fc6"
    top: "fc7"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    inner_product_param {
      num_output: 4096
      weight_filler {
        type: "gaussian"
        std: 0.005
      }
      bias_filler {
        type: "constant"
        value: 0.1
      }
    }
  }
  layer {
    name: "relu7"
    type: "ReLU"
    bottom: "fc7"
    top: "fc7"
  }
  layer { // Dropout 层, 参数含义同上
    name: "drop7"
    type: "Dropout"
  }

```

```

    bottom: "fc7"
    top: "fc7"
    dropout_param {
        dropout_ratio: 0.5
    }
}
layer {                                // 全连接层
    name: "fc8"
    type: "InnerProduct"
    bottom: "fc7"
    top: "fc8"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    inner_product_param {
        num_output: 1000
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}
layer {                                // loss 层
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "fc8"
    bottom: "label"
    top: "loss"
}
I0819 15:18:11.569427 4699 layer_factory.hpp:77] Creating layer data
.....
以下内容省略

```

AlexNet 模型的训练参数, 默认 batch 大小 256, 迭代 450000 次, 约 90 个 epoch。learning rate 初始化为 0.01, 采用 step 的算法, 每 100000 次(约 20 个 epoch)迭代衰减一次。momentum 值为 0.9, weight decay 为 0.0005, 每 10000 个迭代输出一个 snapshot。

6.4.7 Caffe 的 AlexNet 模型与论文的不同

- (1) Caffe 中的 AlexNet 模型没有进行数据增多的操作;
- (2) 用于训练的图片大小不一样, 论文中实际用于训练的图片大小为 224×224 , Caffe 的大小是 227×227 , 而且对训练图片进行了随机镜像;
- (3) non-zero biases 被初始化为 0.1 而不是 1;

6.4.8 ImageNet 模型测试

利用已经训练好的模型对测试数据集的数据生成一个结果, 该结果为一个文本文件, 文件中的每一行对应一张图片, 以图片名称的升序排列, 如从 ILSVRC2012_val_00000001.JPEG 到 ILSVRC2012_val_00050000.JPEG。每一行包含对图片预测的结果标记, 即预测图片所属类别的值 (0 至 999 的整数), 并以 confidence 值的降序排列。每一行的标记数目可以变化, 但不能超过 5。验证数据预测结果的示例文件可见 ILSVRC2012_devkit 中的/evaluation/demo.val.pred.txt

在 caffe/examples 目录下通过 Python 执行 alexnettest.py 文件, 生成结果文件 alexnetlog.txt:

```
$python alexnettest.py
```

alexnettest.py 脚本内容如下:

```
import numpy as np
caffe_root = '../' # this file is expected to be in {caffe_root}/examples
val_dir = '/dataset/imagenet/val'
model_name = 'caffenet_train_iter_450000.caffemodel'
import sys
sys.path.insert(0, caffe_root + 'python')
import caffe
import os
caffe.set_mode_cpu()
net = caffe.Net(caffe_root + 'models/bvlc_reference_caffenet/deploy.prototxt',
               caffe_root + 'models/bvlc_reference_caffenet/' + model_name,
               caffe.TEST)
transformer = caffe.io.Transformer({'data': net.blobs['data'].data.shape})
transformer.set_transpose('data', (2, 0, 1))
```

```

transformer.set_mean('data', np.load(caffe_root
    + 'python/caffe/imagenet/ilsrvr_2012_mean.npy')).mean(1).mean(1))
transformer.set_raw_scale('data', 255) # the reference model operates on
images in [0,255] range instead of [0,1]
transformer.set_channel_swap('data', (2,1,0)) # the reference model has
channels in BGR order instead of RGB
net.blobs['data'].reshape(50,3,227,227)
fh = open(alexnetlog.txt, 'w')
batchsize = net.blobs['data'].shape[0]
for dirpath, dirnames, filenames in os.walk(val_dir):
    sortedfiles = sorted(filenames)
n=len(sortedfiles)
nbatch = (n+ batchsize - 1) // batchsize
for i in range(nbatch):
    idx = np.arange(i*batchsize, min(n, (i+1)*batchsize))
    for tdx in idx:
        filename = sortedfiles[tdx]
        indexofdata= tdx*batchsize
        net.blobs['data'].data[indexofdata]=
            transformer.preprocess('data', caffe.io.load_image
                (os.path.join(dirpath, filename)))

    out = net.forward()
    for j in range(batchsize)
        output_pred=out['prob'][j].argsort() [-1:-6:-1]
        outlist=output_pred.tolist()
        templist=[str(i) for i in outlist]
        fh.write(' '.join(templist))
        fh.write('\n')
fh.close()

```

参数文献

- [1] ImageNet classification with deep convolutional neural networks, Alex Krizhevsky etc, NIPS 2012.

7

第 7 章 GoogLeNet 模型

上一章我们介绍了 CNN 领域内具有重要意义的网络模型 AlexNet，它在 2012 年 ILSVRC 竞赛取得第一名的成绩，具有很多创新点，Top-5 错误率大幅度下降。由于 AlexNet 模型回归到全连接层，参数总量达到 60M。如果增加网络中的隐含层，利用加深神经网络的层数进一步提高网络的识别成功率，不仅会加大网络中参数的总量，使得计算量变得异常庞大，而且极易出现过拟合现象。2014 年的 ILSVRC，Google 也参与此次竞赛，其网络模型为 GoogLeNet。它通过增加模型的层数至 22 层，利用 multi-scale data training，取得第一名的成绩，Top-5 错误率只有 6.66%，但参数只有 7M，远远小于 AlexNet 模型的网络参数。Google 之所以为这一网络模型命名“GoogLeNet”，而非“GoogleNet”，是为了向 LeNet 的作者致敬。Google 的 GooLeNet 模型成功地证明了用更多的卷积，更深的网络层数可以得到更好的预测效果。

本章先介绍 GoogLeNet 模型的背景和 Inception 结构，然后对此模型进行具体解读并分析其特点，最后分析了此模型在 Caffe 的具体实现。

7.1 GoogLeNet 模型简介

GoogLeNet 网络模型是一个 22 层的深度网络，在 <http://arxiv.org/pdf/1409.4842v1.pdf>

(Going deeper with convolutions)^[1]论文中对这一模型有具体的阐述。这个模型证明了用更多的卷积、更深的层次可以得到更好的结果。

7.1.1 背景和动机

神经网络和深度学习技术的快速发展,人们容易通过更高性能的硬件,更庞大的带标签训练数据和更深更宽的网络模型等手段,获得更好的预测识别效果,但这一策略带来两个重要缺陷。

一个缺陷是更深更宽的网络模型会产生巨量参数,从而容易出现过拟合现象。这个问题在标签数据非常有限时特别突出,要想避免过拟合,对少量标签数据的要求和技巧非常高。另一个缺陷是网络规模加大会极大增加计算量,消耗更多的计算资源。实际应用当中,计算资源的预算都是非常有限的,一个非常高效的分布式计算资源对不断增长的网络规模变得越来越重要。

解决这两个缺陷的根本办法是将全连接甚至一般的卷积转化为稀疏连接。一方面现实生物神经系统的连接是稀疏的,另一方面文献^[2]提到,对于大规模稀疏的神经网络,可以通过分析激活值的统计特性和对高度相关的输出进行聚类来逐层构建出一个最优网络。这表明臃肿的稀疏网络可以被不失性能地简化。虽然数学证明有着严格的条件限制,但Hebbian 定理有力地支持这一结论。

由于计算机软硬件对非均匀稀疏数据的计算效率很差,AlexNet 模型重新启用了全连接层,其目的是为了更好地优化并行计算。所以,我们面临这样一个问题,是否有一种方法,既能保持网络结构的稀疏性,又能利用密集矩阵的高计算性能。事实上,可以将稀疏矩阵聚类为较为密集的子矩阵来提高计算性能,Google 团队沿着这一思路提出了 Inception 结构来实现此目标。

7.1.2 Inception 结构

Inception 结构的主要思想是如何找出最优的局部稀疏结构并将其覆盖为近似的稠密组件。文献^[2]提出了一个层与层的结构,结构的最后一层进行相关性统计,将高相关性的聚

集到一起。这些簇构成下一层的单元，与上一层的单元连接。假设上一层的每个单元对应输入图像的某些区域，这些单元被滤波器进行分组。低层的单元集中在某些局部区域，最终会得到在单个区域的大量群，它们能在下一层通过 1×1 卷积覆盖。当然，也可以通过一个簇覆盖更大的空间来减少簇的数量。为了避免块对齐问题，滤波器大小限制在 1×1 、 3×3 和 5×5 。

总的来说，Inception 结构的主要思路是用密集成分来近似最优的局部稀疏结构。Google 团队提出如图 7-1 所示的 Inception 模型 A。

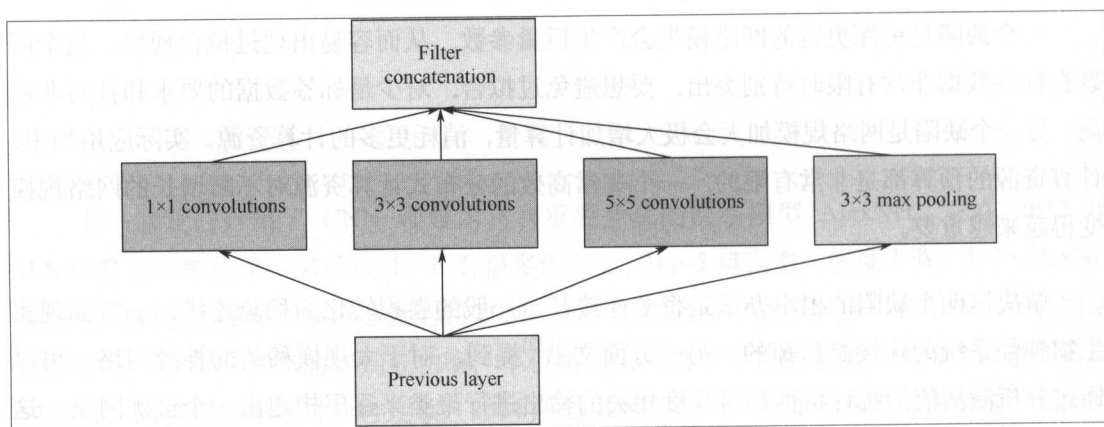


图 7-1 Inception 模型 A

Inception 模型 A 中，卷积核大小不同，感受野的大小也不同，最后的拼接意味着不同尺度性的融合。卷积核大小选择 1、3 和 5，是为了对齐方便。比如在卷积步长为 1 时，只要分别设定 pad 为 0、1 和 2，则卷积后就可以得到相同维度的特征。

网络越到后面，特征越来越抽象，每个特征所涉及的感受野也更大。因此，随着层数的增加， 3×3 和 5×5 卷积的比例也增加。但是，使用 5×5 的卷积核仍然会带来巨大的计算量，必须采用 1×1 的卷积核进行降维，改进后的 Inception 模型 B 如图 7-2 所示。

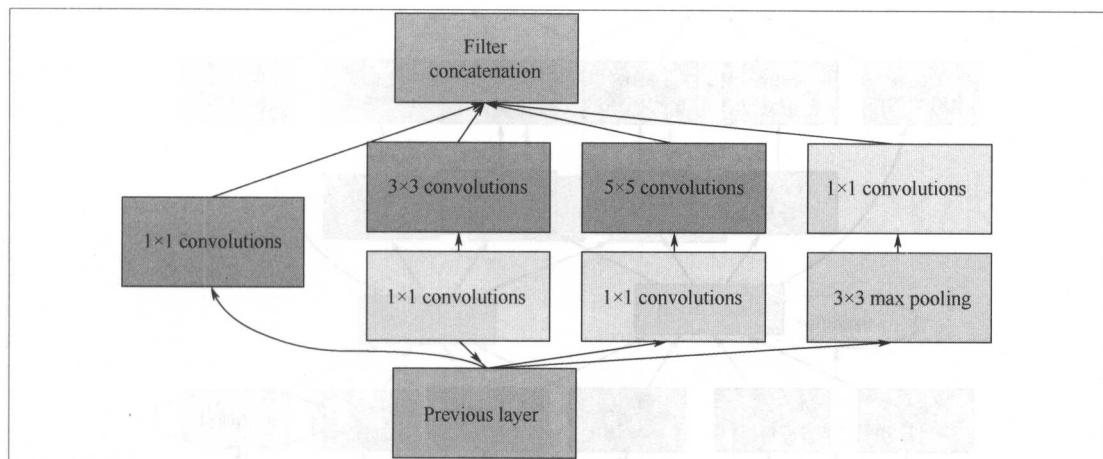
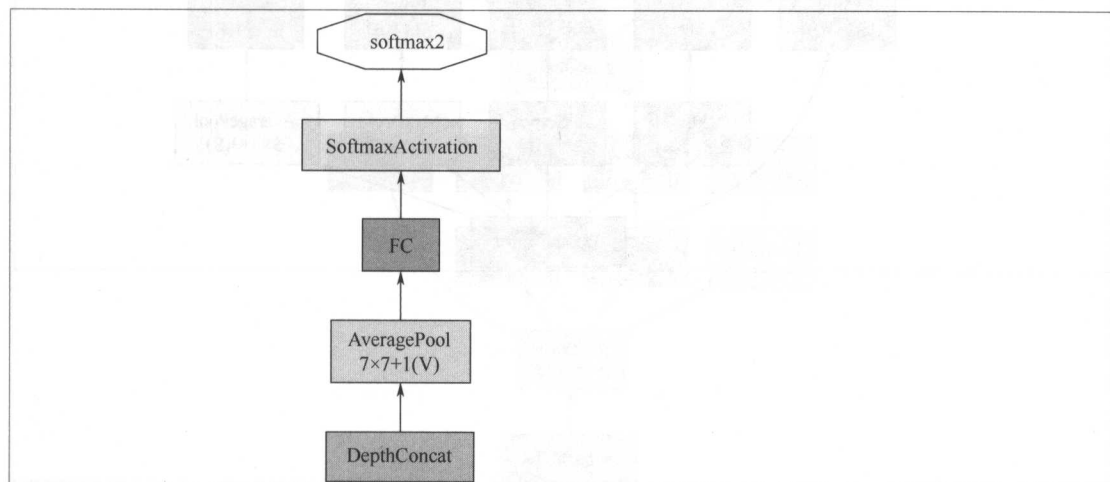


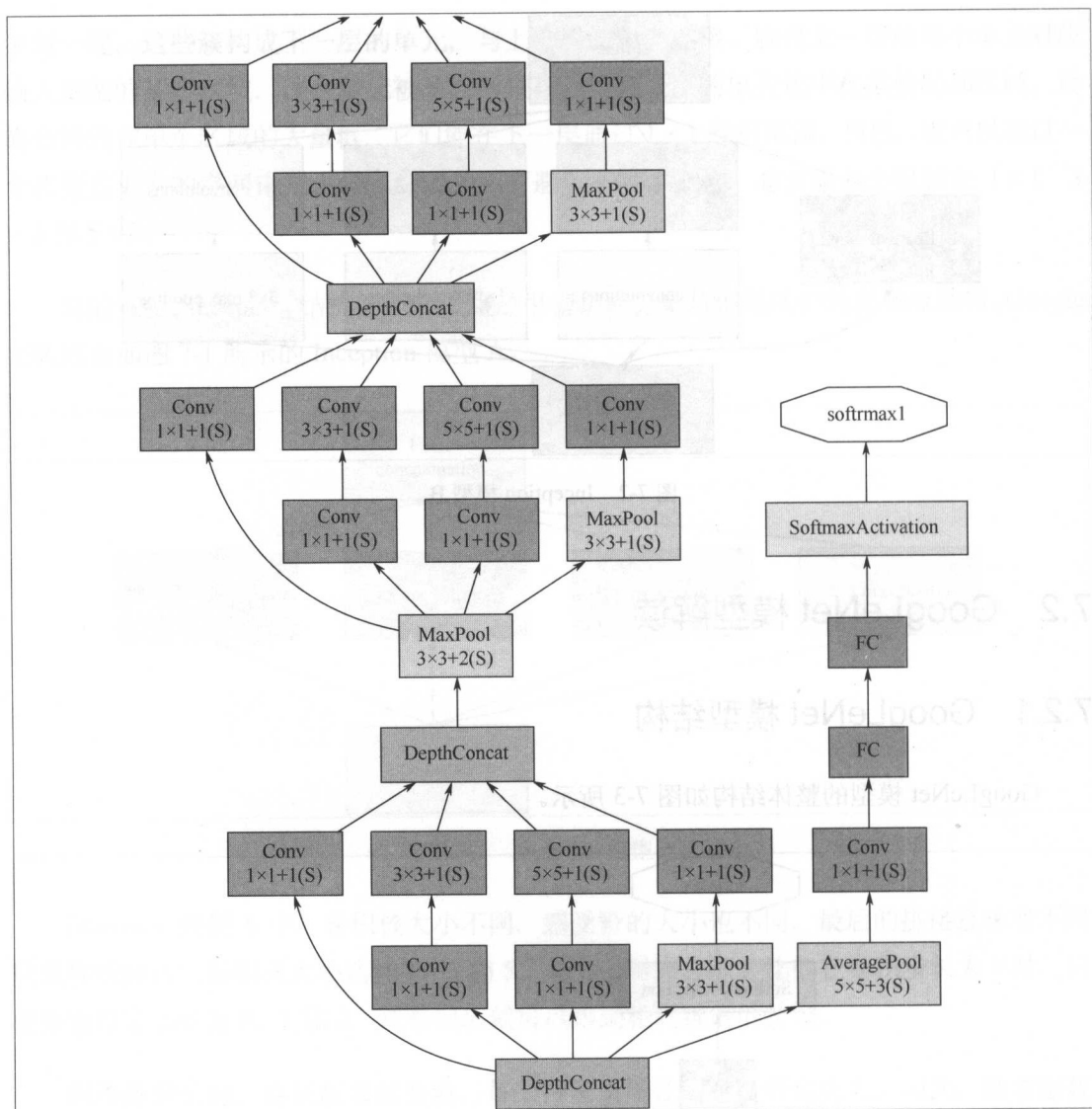
图 7-2 Inception 模型 B

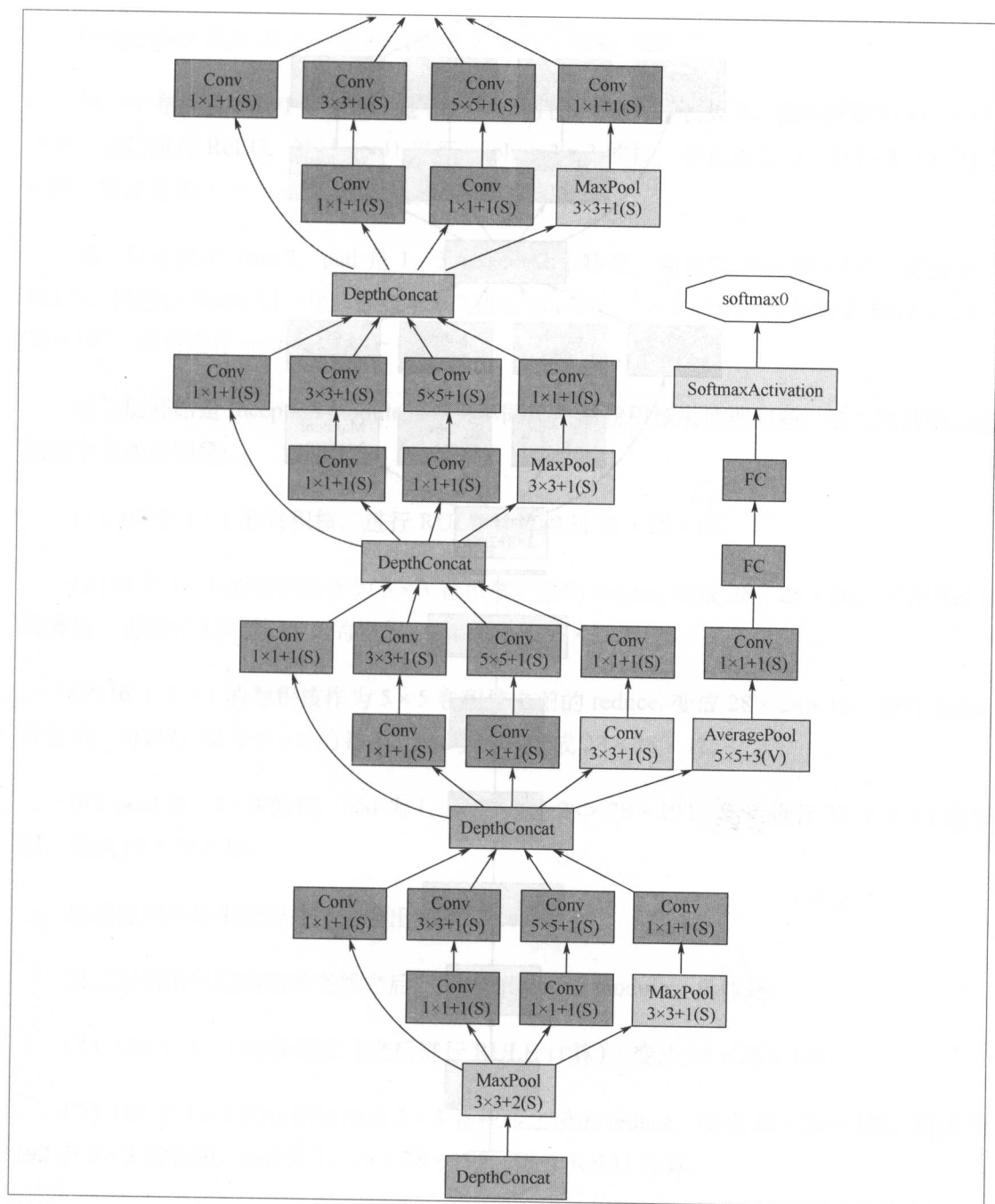
7.2 GoogLeNet 模型解读

7.2.1 GoogLeNet 模型结构

GoogLeNet 模型的整体结构如图 7-3 所示。







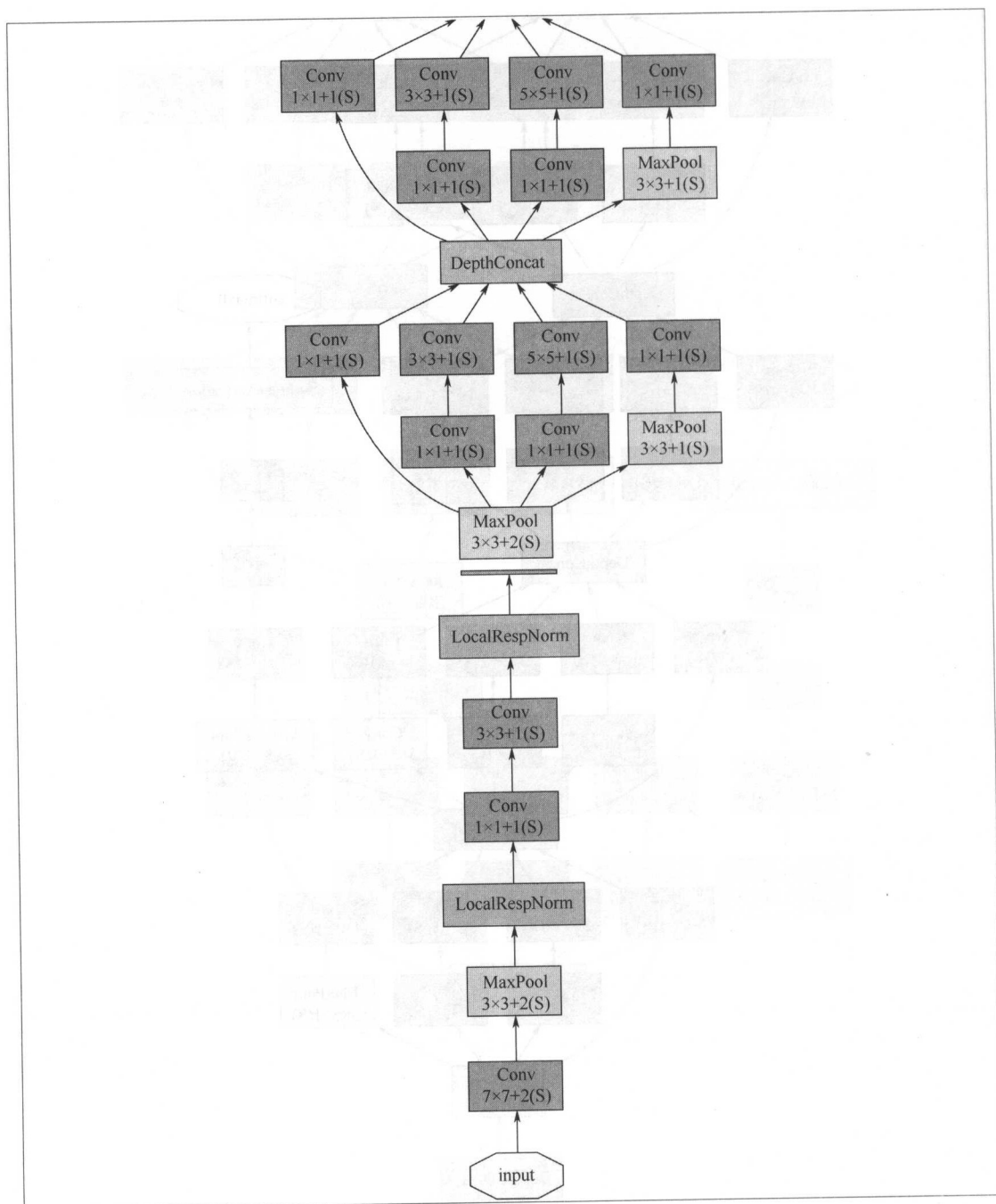


图 7-3 GoogLeNet 模型整体结构

GoogLeNet 共有 22 层，原始数据输入大小为 $224 \times 224 \times 3$ 。

第一个卷积层 conv1 中，pad 是 3，64 个特征， 7×7 步长为 2，输出特征为 $112 \times 112 \times 64$ ，然后进行 ReLU，经过 pool1 进行 pooling 3×3 的核，步长为 2， $\lfloor (112 - 3 + 1) / 2 \rfloor + 1 = 56$ ，特征为 $56 \times 56 \times 64$ ，然后进行 Norm 归一化。

第二层卷积层 conv2，pad 是 1， 3×3 ，192 个特征，输出为 $56 \times 56 \times 192$ ，然后进行 ReLU，再进行 Norm 归一化，经过 pool2 进行 pooling， 3×3 的核，步长为 2 输出为 $28 \times 28 \times 192$ ，然后进行 split 分成四个支线。

第三层开始是 Inception module，采用不同尺度的卷积核来处理问题。第二层到第三层的四个支线分别是：

- (1) 64 个 1×1 的卷积核，进行 RULE 计算得到 $28 \times 28 \times 64$ 。
- (2) 96 个 1×1 的卷积核作为 3×3 卷积核之前的 reduce，变成 $28 \times 28 \times 96$ ，进行 ReLU 计算后，再进行 128 个 3×3 的卷积，pad 为 1， $28 \times 28 \times 128$ 。
- (3) 16 个 1×1 的卷积核作为 5×5 卷积核之前的 reduce，变成 $28 \times 28 \times 16$ ，进行 ReLU 计算后，再进行 32 个 5×5 的卷积，pad 为 2，变成 $28 \times 28 \times 32$ 。
- (4) pool 层， 3×3 的核，pad 为 1，输出还是 $28 \times 28 \times 192$ ，然后进行 32 个 1×1 的卷积，变成 $28 \times 28 \times 32$ 。

最后将四个结果进行连接，输出为 $28 \times 28 \times 256$ 。

第二层到第三层的四条支线之后，开始 Inception module，具体是：

- (1) 128 个 1×1 的卷积核（之后进行 RULE 计算）变成 $28 \times 28 \times 128$ 。
- (2) 128 个 1×1 的卷积核作为 3×3 卷积核之前的 reduce，变成 $28 \times 28 \times 128$ ，再进行 192 个 3×3 的卷积，pad 为 1， $28 \times 28 \times 192$ ，进行 ReLU 计算。
- (3) 32 个 1×1 的卷积核作为 5×5 卷积核之前的 reduce，变成 $28 \times 28 \times 32$ ，进行 ReLU

计算后, 再进行 96 个 5×5 的卷积, pad 为 2, 变成 $28 \times 28 \times 96$ 。

(4) pool 层, 3×3 的核, pad 为 1, 输出还是 $28 \times 28 \times 256$, 然后进行 64 个 1×1 的卷积, 变成 $28 \times 28 \times 64$ 。

最后将四个结果进行连接, 输出为 $28 \times 28 \times 480$ 。

同理依次推算, 数据变化如表 7-1 所示:

表 7-1 GoogLeNet 网络模型参数变化

type	patch_size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception(3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception(3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception(4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception(4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception(4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception(4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception(4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception(5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception(5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout(40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

7.2.2 GoogLeNet 模型特点

GoogLeNet 采用 Inception 结构, 不仅进一步提升了预测分类的成功率, 而且极大地减

少了参数量，分析其原因有如下的特点：

- (1) 采用了模块化的结构，方便增添和修改；
- (2) 网络最后用 average pooling 代替全连接层，将 Top-1 的成功率提高了 0.6%；
- (3) 网络移除了全连接层，但保留了 Dropout 层；
- (4) 网络增加了两个辅助的 softmax 用于向前传导梯度，避免梯度消失。

7.3 GoogLeNet 模型的 Caffe 实现

Caffe 安装后，在 `caffe-master/models/bvlc_googlenet` 目录下有网络的具体定义，内容如下：

```
name: "GoogLeNet"
layer {
    name: "data"
    type: "Data"
    top: "data"
    top: "label"
    include {
        phase: TRAIN // 训练阶段
    }
    transform_param {
        mirror: true
        crop_size: 224
        mean_value: 104
        mean_value: 117
        mean_value: 123
    }
    data_param {
        source: "examples/imagenet/ilsrvrc12_train_lmdb" // 指定训练数据集文件
        batch_size: 32 // batch 大小 32
        backend: LMDB
    }
}
layer {
    name: "data"
    type: "Data"
    top: "data"
    top: "label"
    include {
        phase: TEST // 测试阶段
```

```

    }
    transform_param {
      mirror: false
      crop_size: 224
      mean_value: 104
      mean_value: 117
      mean_value: 123
    }
    data_param {
      source: "examples/imagenet/ilsvrc12_val_lmdb" // 指定测试数据集文件
      batch_size: 50 // batch 大小 50
      backend: LMDB
    }
  }
  layer {
    name: "conv1/7x7_s2" // 卷积层 112*112*64
    type: "Convolution"
    bottom: "data"
    top: "conv1/7x7_s2"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    convolution_param {
      num_output: 64
      pad: 3
      kernel_size: 7 // 卷积核大小
      stride: 2 // 步长 2
      weight_filler {
        type: "xavier"
      }
      bias_filler {
        type: "constant"
        value: 0.2
      }
    }
  }
}
layer { // ReLU 层
  name: "conv1/relu_7x7"
  type: "ReLU"
  bottom: "conv1/7x7_s2"
  top: "conv1/7x7_s2"
}
layer { // Max Pooling 层 56*56*64
  name: "pool1/3x3_s2"
  type: "Pooling"

```



```

    bottom: "conv1/7x7_s2"
    top: "pool1/3x3_s2"
    pooling_param {
        pool: MAX
        kernel_size: 3      // 核大小 3
        stride: 2           // 步长 2
    }
}
layer {                      // LRN 层
    name: "pool1/norm1"
    type: "LRN"
    bottom: "pool1/3x3_s2"
    top: "pool1/norm1"
    lrn_param {
        local_size: 5
        alpha: 0.0001
        beta: 0.75
    }
}
layer {                      // 卷积层
    name: "conv2/3x3_reduce"
    type: "Convolution"
    bottom: "pool1/norm1"
    top: "conv2/3x3_reduce"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 64
        kernel_size: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0.2
        }
    }
}
layer {                      // ReLU 层
    name: "conv2/relu_3x3_reduce"
    type: "ReLU"
    bottom: "conv2/3x3_reduce"
    top: "conv2/3x3_reduce"
}
layer {                      // 卷积层 56*56*192

```



```

name: "conv2/3x3"
type: "Convolution"
bottom: "conv2/3x3_reduce"
top: "conv2/3x3"
param {
  lr_mult: 1
  decay_mult: 1
}
param {
  lr_mult: 2
  decay_mult: 0
}
convolution_param {
  num_output: 192
  pad: 1
  kernel_size: 3
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
    value: 0.2
  }
}
}
layer {
  name: "conv2/relu_3x3"
  type: "ReLU"
  bottom: "conv2/3x3"
  top: "conv2/3x3"
}
layer {
  name: "conv2/norm2"
  type: "LRN"
  bottom: "conv2/3x3"
  top: "conv2/norm2"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
layer { // Max Pooling 层 28*28*192
  name: "pool2/3x3_s2"
  type: "Pooling"
  bottom: "conv2/norm2"
  top: "pool2/3x3_s2"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}

```

```

    }
}
layer {                                // Inception 3a 28*28*256
    name: "inception_3a/1x1"
    type: "Convolution"
    bottom: "pool2/3x3_s2"
    top: "inception_3a/1x1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 64
        kernel_size: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0.2
        }
    }
}
}
..... // 省略 Inception 3a 部分内容
layer {                                // Inception 3b 28*28*480
    name: "inception_3b/1x1"
    type: "Convolution"
    bottom: "inception_3a/output"
    top: "inception_3b/1x1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 128
        kernel_size: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0.2
        }
    }
}

```

```

    }
}
..... // 省略 Inception 3b 部分内容
layer {                                // Max Pooling 层 14*14*480
    name: "pool3/3x3_s2"
    type: "Pooling"
    bottom: "inception_3b/output"
    top: "pool3/3x3_s2"
    pooling_param {
        pool: MAX
        kernel_size: 3
        stride: 2
    }
}
layer {                                // Inception 4a 14*14*512
    name: "inception_4a/1x1"
    type: "Convolution"
    bottom: "pool3/3x3_s2"
    top: "inception_4a/1x1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 192
        kernel_size: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0.2
        }
    }
}
.....// 省略 Inception 4a 部分内容

layer {                                // Inception 4b 14*14*512
    name: "inception_4b/1x1"
    type: "Convolution"
    bottom: "inception_4a/output"
    top: "inception_4b/1x1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {

```



```

    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 160
    kernel_size: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
      value: 0.2
    }
  }
}
.....// 省略 Inception 4b 部分内容
layer {                                // Inception 4c 14*14*512
  name: "inception_4c/1x1"
  type: "Convolution"
  bottom: "inception_4b/output"
  top: "inception_4c/1x1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 128
    kernel_size: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
      value: 0.2
    }
  }
}
.....// 省略 Inception 4c 部分内容
layer {                                // Inception 4d 14*14*528
  name: "inception_4d/1x1"
  type: "Convolution"
  bottom: "inception_4c/output"
  top: "inception_4d/1x1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
}

```

```

param {
    lr_mult: 2
    decay_mult: 0
}
convolution_param {
    num_output: 112
    kernel_size: 1
    weight_filler {
        type: "xavier"
    }
    bias_filler {
        type: "constant"
        value: 0.2
    }
}
}
}
.....// 省略 Inception 4d 部分内容
layer {                                // Inception 4e 14*14*832
    name: "inception_4e/1x1"
    type: "Convolution"
    bottom: "inception_4d/output"
    top: "inception_4e/1x1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256
        kernel_size: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0.2
        }
    }
}
}
.....// 省略 Inception 4e 部分内容
layer {                                // Max Pooling 层 7*7*832
    name: "pool4/3x3_s2"
    type: "Pooling"
    bottom: "inception_4e/output"
    top: "pool4/3x3_s2"
    pooling_param {
        pool: MAX
        kernel_size: 3
    }
}

```

```

        stride: 2
    }
}
layer {                                // Inception 5a 7*7*832
    name: "inception_5a/1x1"
    type: "Convolution"
    bottom: "pool4/3x3_s2"
    top: "inception_5a/1x1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256
        kernel_size: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0.2
        }
    }
}
}
.....// 省略 Inception 5a 部分内容
layer {                                // Inception 5b 7*7*1024
    name: "inception_5b/relu_1x1"
    type: "ReLU"
    bottom: "inception_5b/1x1"
    top: "inception_5b/1x1"
}
.....// 省略 Inception 5b 部分内容
layer {                                // Average Pooling 层 1*1*1024
    name: "pool5/7x7_s1"
    type: "Pooling"
    bottom: "inception_5b/output"
    top: "pool5/7x7_s1"
    pooling_param {
        pool: AVE
        kernel_size: 7
        stride: 1
    }
}
}
layer {                                // Dropout 层 1*1*1024
    name: "pool5/drop_7x7_s1"
    type: "Dropout"
    bottom: "pool5/7x7_s1"

```



```

    top: "pool5/7x7_s1"
    dropout_param {
      dropout_ratio: 0.4
    }
  }
  layer {                                // 全连接层 1*1*1000
    name: "loss3/classifier"
    type: "InnerProduct"
    bottom: "pool5/7x7_s1"
    top: "loss3/classifier"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    inner_product_param {
      num_output: 1000
      weight_filler {
        type: "xavier"
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
}
layer {                                // SoftmaxWithLoss 层 1*1*1000
  name: "loss3/loss3"
  type: "SoftmaxWithLoss"
  bottom: "loss3/classifier"
  bottom: "label"
  top: "loss3/loss3"
  loss_weight: 1
}
layer {
  name: "loss3/top-1"
  type: "Accuracy"
  bottom: "loss3/classifier"
  bottom: "label"
  top: "loss3/top-1"
  include {
    phase: TEST
  }
}
layer {
  name: "loss3/top-5"
  type: "Accuracy"
  bottom: "loss3/classifier"

```

```

bottom: "label"
top: "loss3/top-5"
include {
  phase: TEST
}
accuracy_param {
  top_k: 5
}
}

```

由于 GoogLeNet 的训练计算量比较大, 我们可以从 http://dl.caffe.berkeleyvision.org/bvlc_googlenet.caffemodel 下载训练好的模型用于识别和分类。

参数文献

- [1] Christian Szegedy, etc. Going Deeper with Convolutions 17 Sep 2014.
- [2] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning some deep representations. CoRR, abs/1310.6343, 2013

8

第 8 章 VGGNet 模型

上一章我们介绍了取得 2014 年 ILSVRC 竞赛第一名的 GoogLeNet 模型，它采用更深的网络结构，更少的参数量，取得了 Top-5 错误率只有 6.66% 的巨大成功。2014 年 ILSVRC 竞赛的第二名是 VGGNet 网络模型，Top-5 错误率为 7.3%。VGGNet 继承了 LeNet 和 AlexNet 的框架，采用了一个 19 层的深度网络。虽然 VGGNet 模型在分类成功率与 GoogLeNet 相比，稍现逊色，但在多个迁移学习任务中的表现要优于 GoogLeNet。而且，从图像中提取 CNN 特征，VGGNet 模型是首选算法。VGGNet 的缺点是需要更大的存储空间，参数量达到 140M。因此，研究和学习 VGGNet 这一经典网络模型仍然有很高的价值。

本章先介绍 VGGNet 网络模型的特点，然后对其进行了进一步的解读，最后介绍了 VGGNet 的训练方法、模型分类和定位，以及整个网络 Caffe 实现。

8.1 VGGNet 网络模型

8.1.1 VGGNet 模型介绍

VGGNet 是牛津大学 VGG (Visual Geometry Group) 视觉几何组 Karen Simonyan 和 Andrew Zisserman 于 2014 年撰写的论文^[1]中提出的卷积神经网络结构，VGGNet 建立了一

个 19 层深度网络，在 ILSVRC 取得了定位第一、分类第二的成绩。VGGNet 网络模型与 AlexNet 框架有很多相似之处，有五个 Group 的卷积，二层 FC 图像特征，一层 FC 分类特征。

VGGNet 是从 AlexNet 发展而来的，主要进行了两个方面的改进：（1）在第一卷积层使用更小的 filter 尺寸和间隔。（2）在整个图片和 multi-scale 上训练和测试图片。

8.1.2 VGGNet 模型特点

VGGNet 与 AlexNet 相比，除了使用更多的层之外，VGGNet 所有的卷积层都是同样大小的 filter，尺寸为 3×3 ，卷积间隔 $S=1$ ，并且 3×3 的卷积层有一个像素的填充。VGGNet 网络模型的特点如下：

- （1） 3×3 是最小的能够捕获上下左右和中心概念的尺寸。
- （2）两个 3×3 的卷积层是 5×5 ，三个 3×3 是 7×7 ，可以替代大的 filter 尺寸。
- （3）多个 3×3 的卷积层比一个大尺寸 filter 卷积层有更多的非线性，使得判决函数更加具有判决性。
- （4）多个 3×3 的卷积层比一个大尺寸的 filter 有更少的参数。

8.1.3 VGGNet 模型解读

VGGNet 的网络结构如图 8-1 所示。

VGGNet 有五个 max-pooling 层，故是五阶段卷积特征提取，每层的卷积个数从首阶段的 64 个开始，每个阶段增长一倍，直到达到最高的 512 个，然后保持不变。

结构 A: Input (224, 224, 3) \rightarrow 64F (3, 3, 3, 1) \rightarrow max-p(2, 2) \rightarrow 128F (3, 3, 64, 1) \rightarrow max-p(2, 2) \rightarrow 256F (3, 3, 128, 1) \rightarrow 256F (3, 3, 256, 1) \rightarrow max-p(2, 2) \rightarrow 512F (3, 3, 256, 1) \rightarrow 512F (3, 3, 512, 1) \rightarrow max-p(2, 2) \rightarrow 512F (3, 3, 256, 1) \rightarrow 512F (3, 3, 512, 1) \rightarrow max-p(2, 2) \rightarrow 4096fc \rightarrow 4096fc \rightarrow 1000softmax。有 8 个卷积层，3 个全连

接层，共计 11 层。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input(224×224 RGB image)					
conv3-64	conv3-6 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 8-1 VGGNet 网络结构

结构 B: 在 A 的 stage2 和 stage3 分别增加一个 3×3 的卷积层，有 10 个卷积层，总计 13 层。

结构 C: 在 B 的基础上，stage3，stage4，stage5 分别增加 1×1 的卷积层，有 13 个卷

积层，总计 16 层。

结构 D：在 C 的基础上，stage3，stage4，stage5 分别增加 3×3 的卷积层，有 13 个卷积层，总计 16 层。

结构 E：在 D 的基础上，stage3，stage4，stage5 分别增加 3×3 的卷积层，有 16 个卷积层，总计 19 层。

各层的网络参数量如下所示。

网络结构	A,A-LRN	B	C	D	E
参数数量	133M	133M	134M	138M	144M

8.2 VGGNet 网络训练

8.2.1 VGGNet 训练参数设置

VGGNet 的最小 batch 值为 256，其他参数不变。虽然 VGGNet 的网络参数比 AlexNet 多，网络层数也多，但 VGGNet 只需要很少的迭代次数就能收敛，这得益于以下两个方法：

- (1) 更深的网络和较小的 filter 尺寸具有隐式规则的作用；
- (2) 先训练浅层网络。在得到 A 网络的参数后，训练更深的网络 E 时，使用 A 中得到的参数初始化对应的层。

8.2.2 Multi-Scale 训练

首先将原始图片等比例缩放，保证图片短边大于 224，然后在图片上随机提取 224×224 窗口，进行训练。由于物体尺度变化多样，Multi-Scale（多尺度）可以更好地识别物体，有两种多尺度的训练方法。

方法 1：在不同的尺度下，训练多个分类器。参数 S 是在原始图片上缩放时的短边长

度。然后训练 $S=256$ 和 $S=384$ 两个分类器, 其中 $S=384$ 的分类器的参数使用 $S=256$ 的参数进行初始化, 且将步长调为 $10e-3$ 。

方法 2: 另一种方法是直接训练一个分类器, 每次数据输入时, 每张图片被重新缩放, 缩放的短边 S 随机从 $[S_{\min}, S_{\max}]$ 中选择 (一般取 $S_{\min}=256$, $S_{\max}=512$), 网络参数初始化时使用 $S=384$ 时的参数。

8.2.3 测试

测试步骤如下:

(1) 首先进行等比例缩放, 短边长度 Q 大于 224, Q 的意义与 S 相同, 只不过 S 是训练数据集中的参数, 而 Q 是测试数据集中的参数。 Q 不必等于 S , 相反, 对于一个 S , 使用多个 Q 值进行测试, 然后去平均会使效果变好。

(2) 将全连接层转换为卷积层, 第一个全连接转换为 7×7 的卷积, 第二个转换为 1×1 的卷积。

8.2.4 部署

利用 C++ Caffe toolbox, 在 4 个 Titan GPU 上并行计算, 比单独 GPU 快 3.75 倍, 每个网络差不多 2~3 周。

8.3 VGGNet 模型分类实验

8.3.1 Single-scale 对比

VGGNet Single-scale 对比如表 8-1 所示。

表 8-1 VGGNet Single-scale 对比

ConvNet	smallest image side		top-1 val.error(%)	top=5 val.error(%)
	train(S)	test(Q)		
A	256	256	29.6	10.4
A-LRN	256	256	29.7	10.5
B	256	256	28.7	9.9
C	256	256	28.1	9.4
	384	384	28.1	9.3
	[256;512]	384	27.3	8.8
D	256	256	27.0	8.8
	384	384	26.8	8.7
	[256;512]	384	25.6	8.1
E	256	256	27.3	9.0
	384	384	26.9	8.7
	[256;512]	384	25.5	8.0

- A 与 A-LRN 比较: A-LRN 结果没有 A 好, 说明 LRN 作用不大。
- A 与 B, C, D, E 比较: 越深越好。
- A 与 C 比较: 增加 1×1 filter, 即增加额外的非线性提升效果。
- C 与 D 比较: 3×3 的 filter 比 1×1 filter 要好, 使用较大的 filter 能够捕捉更大的空间特征。

训练方法: 在 scale 区间[256, 512]通过 scale 增益来训练网络, 比在固定的两个 $S=256$ 和 $S=512$, 结果明显提升。Multi-scale 训练确实很有用, 因为卷积网络对于缩放有一定的不变性, 通过 multi-scale 训练可以增加这种不变性的能力。

8.3.2 Multi-scale 对比

测试数据集是多尺度, 尺度差异过大会导致性能下降, 故测试数据集尺度 Q 和 S 的值

在上下 32 内浮动。当训练数据集是区间尺度时，测试数据集尺度为区间的最小值、最大值和中值。具体如表 8-2 所示。

表 8-2 VGGNet Multi-scale 对比

ConvNet	smallest image side		top-1 val.error(%)	top=5 val.error(%)
	train(S)	test(Q)		
B	256	224,256,288	28.2	9.6
C	256	224,256,288	27.7	9.2
	384	352,384,416	27.8	9.2
	[256;512]	256,384,512	26.3	8.2
D	256	224,256,288	26.6	8.6
	384	352,384,416	26.5	8.6
	[256;512]	256,384,512	24.8	7.5
E	256	224,256,288	26.9	8.7
	384	352,384,416	26.7	8.6
	[256;512]	256,384,512	24.8	7.5

8.3.3 模型融合

模型融合的方法是取后验概率估计的均值，如表 8-3。融合表 8-1 和 8-2 中两个最好的 D 和 E 模型可达到更好的值，而融合七个模型则会变差。

表 8-3 多模型融合对比

Combined ConvNet models	Error		
	top-1 val	top-5 val	top-5 test
(D/[256;512]/256,384,512),(E/[256;512]/256,384,512)	24.0	7.1	7.0
(D/256/224,256,288),(D/384/352,384,416),(D/[256;512]/256,384,512) (C/256/224,256,288),(C/384/352,384,416) (E/256/224,256,288),(E/384/352,384,416)	24.7	7.5	7.3

8.4 VGGNet 网络结构

VGGNet 的网络结构可参考 http://cs.stanford.edu/people/karpathy/vgg_train_val.prototxt^[2] 文件, 文件内容如下:

```
name: "VGG_ILSVRC_16_layers"
layers {
    name: "data"                // data 层
    type: DATA
    include {
        phase: TRAIN           // 训练阶段
    }
    transform_param {           // 图片变换参数
        crop_size: 224
        mean_value: 104
        mean_value: 117
        mean_value: 123
        mirror: true
    }
    data_param {
        source: "data/ilsvrc12/ilsvrc12_train_lmdb" // 训练数据集
        batch_size: 64                               // batch 大小
        backend: LMDB
    }
    top: "data"
    top: "label"
}
layers {
    name: "data"
    type: DATA
    include {
        phase: TEST           // 测试阶段
    }
    transform_param {
        crop_size: 224
        mean_value: 104
        mean_value: 117
        mean_value: 123
        mirror: false
    }
    data_param {
        source: "data/ilsvrc12/ilsvrc12_val_lmdb"
        batch_size: 50
        backend: LMDB
    }
    top: "data"
    top: "label"
}
```

```

layers {                                     // 卷积层
    bottom: "data"
    top: "conv1_1"
    name: "conv1_1"
    type: CONVOLUTION
    convolution_param {                     // 卷积参数, 填充 1, 卷积核大小为 3
        num_output: 64
        pad: 1
        kernel_size: 3
    }
    blobs_lr: 0
    blobs_lr: 0
}
layers {                                     // ReLU 层
    bottom: "conv1_1"
    top: "conv1_1"
    name: "relu1_1"
    type: RELU
}
layers {                                     // 卷积层, 同上
    bottom: "conv1_1"
    top: "conv1_2"
    name: "conv1_2"
    type: CONVOLUTION
    convolution_param {
        num_output: 64
        pad: 1
        kernel_size: 3
    }
    blobs_lr: 0
    blobs_lr: 0
}
layers {                                     // ReLU 层
    bottom: "conv1_2"
    top: "conv1_2"
    name: "relu1_2"
    type: RELU
}
layers {                                     // MAX Pooling 层
    bottom: "conv1_2"
    top: "pool1"
    name: "pool1"
    type: POOLING
    pooling_param {                       // Pooling 参数, 步长为 2
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
layers {                                     // 卷积层, output 为 128

```



```

    bottom: "pool1"
    top: "conv2_1"
    name: "conv2_1"
    type: CONVOLUTION
    convolution_param {
      num_output: 128
      pad: 1
      kernel_size: 3
    }
    blobs_lr: 0
    blobs_lr: 0
  }
  layers {                                // ReLU 层
    bottom: "conv2_1"
    top: "conv2_1"
    name: "relu2_1"
    type: RELU
  }
  layers {                                // 卷积层, output 为 128
    bottom: "conv2_1"
    top: "conv2_2"
    name: "conv2_2"
    type: CONVOLUTION
    convolution_param {
      num_output: 128
      pad: 1
      kernel_size: 3
    }
    blobs_lr: 0
    blobs_lr: 0
  }
  .....// 相同层结构, 省略
  layers {                                // Dropout 层
    bottom: "fc6"
    top: "fc6"
    name: "drop6"
    type: DROPOUT
    dropout_param {
      dropout_ratio: 0.5    // 丢弃率 0.5
    }
  }
  layers {                                // 全连接层 1
    bottom: "fc6"
    top: "fc7"
    name: "fc7"
    type: INNER_PRODUCT
    inner_product_param {
      num_output: 4096
    }
    blobs_lr: 0
    blobs_lr: 0
  }

```



```

}
layers {
  bottom: "fc7"
  top: "fc7"
  name: "relu7"
  type: RELU
}
layers {
  bottom: "fc7"
  top: "fc7"
  name: "drop7"
  type: DROPOUT
  dropout_param {
    dropout_ratio: 0.5
  }
}
layers {                                // 全连接层 2
  name: "fc8"
  bottom: "fc7"
  top: "fc8"
  type: INNER_PRODUCT
  inner_product_param {
    num_output: 1000
  }
  blobs_lr: 0
  blobs_lr: 0
}
layers {                                // softmax_loss 层
  name: "loss"
  type: SOFTMAX_LOSS
  bottom: "fc8"
  bottom: "label"
  top: "loss/loss"
}
layers {
  name: "accuracy/top1"
  type: ACCURACY
  bottom: "fc8"
  bottom: "label"
  top: "accuracy@1"
  include: { phase: TEST }
  accuracy_param {
    top_k: 1
  }
}
layers {
  name: "accuracy/top5"
  type: ACCURACY
  bottom: "fc8"
  bottom: "label"
  top: "accuracy@5"
}

```

```
include: { phase: TEST }
accuracy_param {
  top_k: 5
}
```

参考文献

- [1] Very Deep Convolutional Networks for Large-Scale Image Recognition, Karen Simonyan, Andrew Zisserman, 2014.9.4
- [2] http://cs.stanford.edu/people/karpathy/vgg_train_val.prototxt

9

第 9 章 Siamese 模型

通过前面几章经典模型的学习，我们对深度学习的算法和模型有了深刻的理解和认识，这些神经网络模型不仅在 ILSVRC 竞赛中取得成功，在图像识别、分类预测等实际应用中也发挥了巨大的作用，比如当前汽车牌照自动识别系统就是基于 LeNet 神经网络模型进一步改进应用到我们的日常生活当中的，Google 的人工智能围棋程序 AlphaGo 也是基于深度卷积神经网络模型的。这些应用都是先基于海量标签数据被机器训练和学习后，然后对未知图片、图像的自动识别和分类。在我们的生活中，会面临这样一类问题：给定两张相似的图片，如何判断它们是“一样的”？更实际的例子是如何判断两份签名是否来自同一人之手？

本章介绍的 Siamese 网络（孪生网络）就是为了解决这样一类现实中的问题而提出的网络模型。在本章里，我们先阐述 Siamese 模型的基本理论，然后解读其 Caffe 的实现，最后介绍 Caffe 环境下的训练方法。

9.1 Siamese 网络模型

9.1.1 Siamese 模型原理

1993 年 Jane Bromley, Isabelle Guyon, Yann LeCun 等人在论文 “Signature verification using a “Siamese” time delay neural network”^[1] 提出用 Siamese 网络模型进行手写签字识别。它的特点是同时接收两个图片输入。这一算法的原理是利用神经网络提取描述算子，得到特征向量，然后利用两个图片的特征向量判断相似度。与 SHIF (Scale Invariant Feature Transform) 不同的是，Siamese 模型利用 CNN 进行特征提取，并且用特征向量构造损失函数，然后训练网络。

Siamese 模型的原理可以用 “Learning a similarity metric discriminatively, with application to face verification” 文献^[2] 的描述来理解，它主要利用了 Siamese 网络进行人脸相似度判断，即人脸识别。Siamese 模型的原理如图 9-1 所示。有两个相同参数，权重共享的 CNN 分支 X_1 和 X_2 ，两张图片分别输入 X_1 和 X_2 ，各自得到一个输出特征向量 $G_w(X_1)$ 和 $G_w(X_2)$ 。接着构造两个特征向量的距离度量作为两张图片的相似度计算函数：

$$E_w(X_1, X_2) = \|G_w(X_1) - G_w(X_2)\|$$

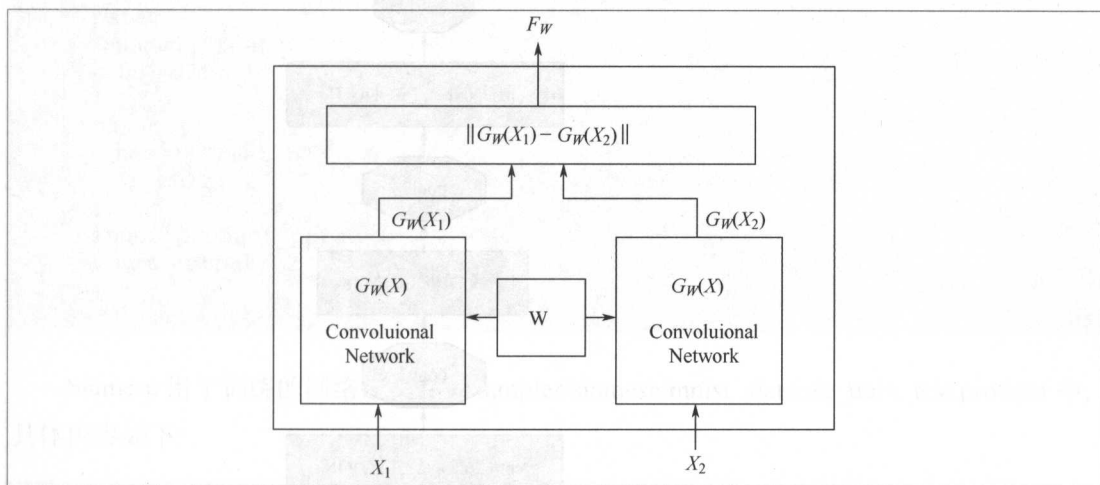


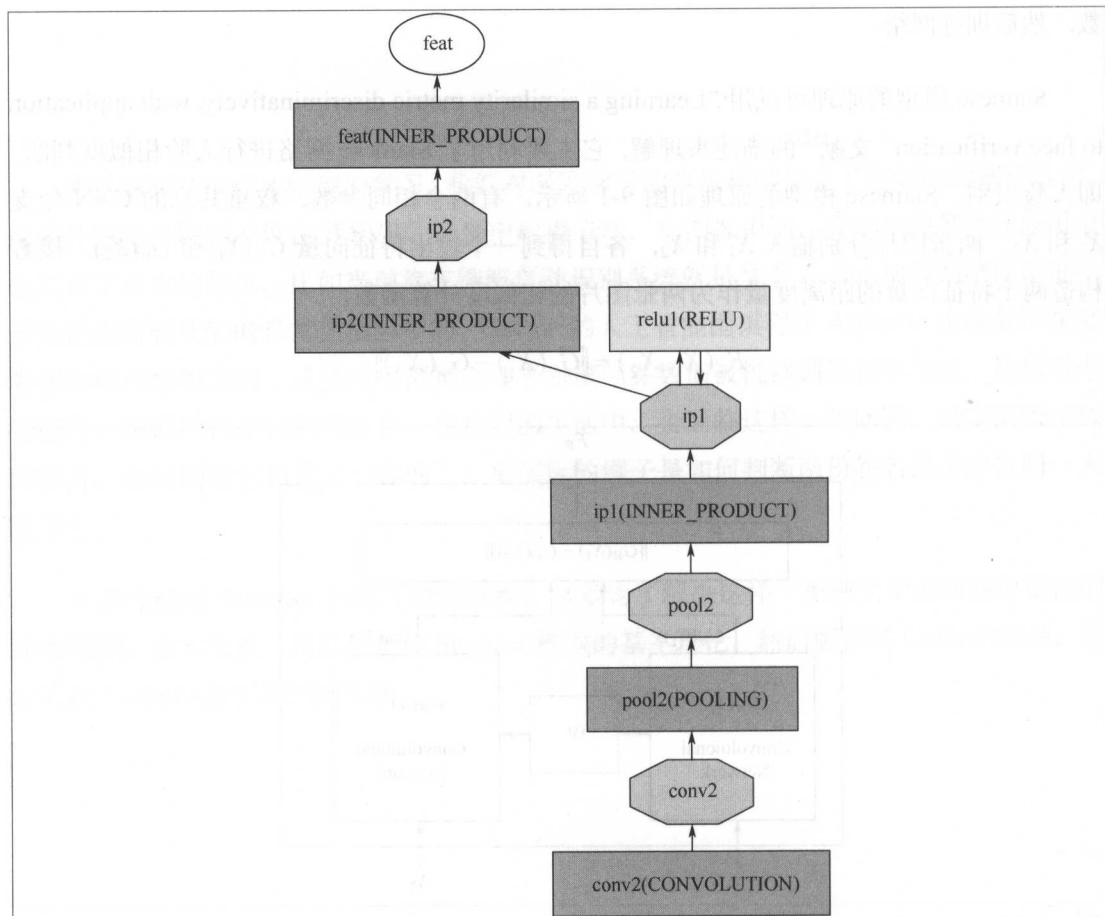
图 9-1 Siamese 模型的原理

$G_w(X_1)$ 和 $G_w(X_2)$ 是以 X_1 和 X_2 为参数的自变量网络映射函数, 是用于评价是否相似的特征向量。基于此定义损失函数, 对网络进行训练, 就可以判别两张人脸的相似度。

因此, Siamese 模型其实是一个提取图片的特征算子的过程, 网络的最后一层定义了特征向量间相似度的损失函数。

9.1.2 Siamese 模型实现

Siamese 模型的 Caffe 实现如图 9-2 所示。



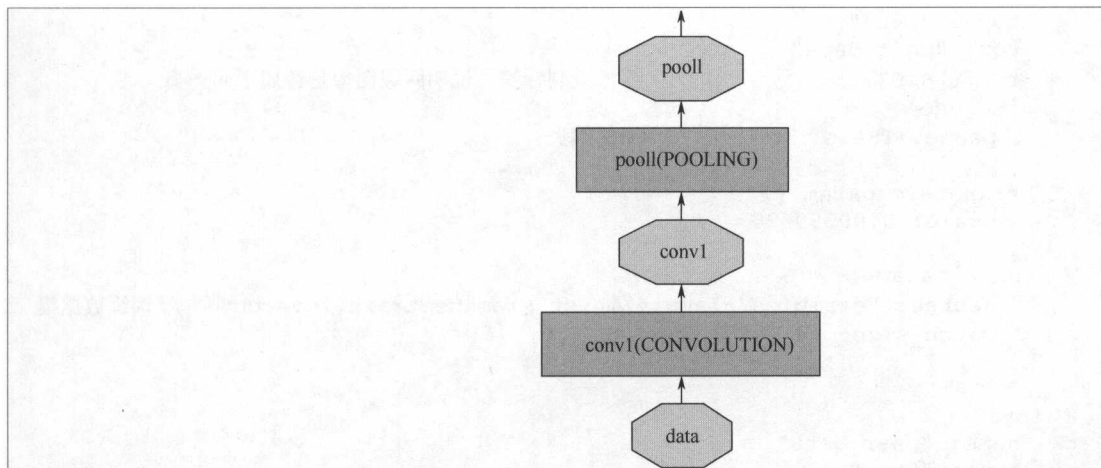


图 9-2 Siamese 模型的 Caffe 实现

Siamese 的卷积网络定义在 `./examples/siamese/mnist_siamese.prototxt` 有具体描述，此网络与 LeNet 模型基本一样，唯一的不同是把生成 10 个数字类别概率的顶层替换为生成二维向量的“特征”层，如下所示：

```

layer {
  name: "feat"
  type: "InnerProduct"
  bottom: "ip2"
  top: "feat"
  param {
    name: "feat_w"
    lr_mult: 1
  }
  param {
    name: "feat_b"
    lr_mult: 2
  }
  inner_product_param {
    num_output: 2
  }
}

```

Siamese 用于训练的网络定义在 `./examples/siamese/mnist_siamese_train_test.prototxt` 中，具体内容如下：

```

name: "mnist_siamese_train_test"
layer {
  name: "pair_data"
  // data 层,
  // 图像数据成对输入
}

```



```

type: "Data"
top: "pair_data"
top: "sim" // 二进制标签, 说明两幅图像是否属于同一类
include {
  phase: TRAIN // 训练阶段
}
transform_param {
  scale: 0.00390625
}
data_param {
  source: "examples/siamese/mnist_siamese_train_leveldb" // 训练数据集
  batch_size: 64
}
}
layer {
  name: "pair_data"
  type: "Data"
  top: "pair_data"
  top: "sim"
  include {
    phase: TEST // 测试阶段
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/siamese/mnist_siamese_test_leveldb"
    batch_size: 100
  }
}
}
layer { // slice 层, 将两个图片分开, 分别做卷积并提取特征
  name: "slice_pair"
  type: "Slice"
  bottom: "pair_data"
  top: "data"
  top: "data_p"
  slice_param {
    slice_dim: 1
    slice_point: 1
  }
}
}
layer { // 卷积层, MAX Pooling 层, 全连接层, ReLU
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    name: "conv1_w"
    lr_mult: 1
  }
  param {

```

```

    name: "conv1_b"
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
... // 省略部分层定义
layer {
  name: "feat"
  type: "InnerProduct"
  bottom: "ip2"
  top: "feat"           // 第一幅图撮的特征
  param {
    name: "feat_w"
    lr_mult: 1
  }
  param {
    name: "feat_b"
    lr_mult: 2
  }
  inner_product_param {
    num_output: 2       // 特征维度为 2
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {                 // data_p 的卷积层和全连接层
  name: "conv1_p"
  type: "Convolution"
  bottom: "data_p"
  top: "conv1_p"
  param {
    name: "conv1_w"
    lr_mult: 1
  }
  param {
    name: "conv1_b"
    lr_mult: 2
  }
}

```

```

    }
    convolution_param {
      num_output: 20
      kernel_size: 5
      stride: 1
      weight_filler {
        type: "xavier"
      }
      bias_filler {
        type: "constant"
      }
    }
  }
}
... // 省略部分层定义
layer {
  name: "feat_p"
  type: "InnerProduct"
  bottom: "ip2_p"
  top: "feat_p"
  param {
    name: "feat_w"
    lr_mult: 1
  }
  param {
    name: "feat_b"
    lr_mult: 2
  }
  inner_product_param {
    num_output: 2
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "loss"
  type: "ContrastiveLoss"
  bottom: "feat"
  bottom: "feat_p"
  bottom: "sim"
  top: "loss"
  contrastive_loss_param {
    margin: 1
  }
}
}

```

Siamese 网络从读取 LevelDB 数据库的 data 层开始，数据库中的每个条目包含的图像

数据为一对图像 (pair_data) 和一个二进制标签, 说明它们是属于同一类还是不同的类。

slice 层把一对图像打包到数据库的同一个 blob 中, 每个通道打包一个图像。因为需要分别使用这两个图像, 所以在 data 层后面加一个 slice 层。这一层负责把 pair_data 按通道切分, 这样在 data 中只有一个图像, 与之成对的图像在 data_p 中。

9.2 Siamese 网络训练

Siamese 网络的训练在 Caffe Tutorial 有具体介绍^[3]。

9.2.1 数据准备

首先从 MNIST 网站上下载并转换数据格式, 在 Linux 下执行以下命令:

```
$/data/mnist/get_mnist.sh
$/examples/siamese/create_mnist_siamese.sh
```

运行后会自动从 MNIST 网站下载数据, 并在 /examples/siamese 目录下生成两个数据库文件 mnist_siamese_train_leveldb 和 examples/siamese/mnist_siamese_test_leveldb。

9.2.2 生成 side

指定 Siamese 网络的第一个 side, 这个 side 在 data 上运行, 生成 feat。在 mnist_siamese.prototxt 网络定义中增加默认的权重配置, 然后为卷积层和全连接层的参数命名, 这样使得 Caffe 能在 Siamese 网络的两个 side 间共享参数, 如下是网络定义中的格式:

```
param { name: "conv1_w" ... }
param { name: "conv1_b" ... }
...
param { name: "conv2_w" ... }
param { name: "conv2_b" ... }
...
param { name: "ip1_w" ... }
param { name: "ip1_b" ... }
...
param { name: "ip2_w" ... }
```



```
param { name: "ip2_b" ... }
...
```

接下来创建第二个 side，其方法和第一个 side 是一样的，只需要改一下每个层输入和输出的名字，然后在名称后加 “_p” 表示“成对”层。

9.2.3 对比损失函数

为了训练网络，需要优化对比损失函数，使得在特征空间里匹配的图像对离得近，不匹配的图像对离得远。这个损失函数由 CONTRASTIVE_LOSS 层实现：

```
layer {
  name: "loss"
  type: "ContrastiveLoss"
  contrastive_loss_param {
    margin: 1.0
  }
  bottom: "feat"
  bottom: "feat_p"
  bottom: "sim"
  top: "loss"
}
```

9.2.4 定义 solver

Siamese 的 solver 文件定义在 examples/siamese/mnist_siamese_solver.prototxt 中，此文指出了网络模型文件。

9.2.5 网络训练

在 Linux 命令行，执行以下命令进行 Siamese 网络模型的训练：

```
./examples/siamese/train_mnist_siamese.sh
```

执行命令后，部分 Log 输出如下：

```
I0917 14:20:01.309239 19415 solver.cpp:486] Iteration 49900, lr = 0.00261174
I0917 14:20:02.473892 19415 solver.cpp:361] Snapshotting to examples/
siamese/mnist_siamese_iter_50000.caffemodel
I0917 14:20:02.483991 19415 solver.cpp:369] Snapshotting solver state to
```

```

examples/siamese/mnist_siamese_iter_50000.solverstate
I0917 14:20:02.484702 19415 solver.cpp:276] Iteration 50000, loss =
0.000978595
I0917 14:20:02.484742 19415 solver.cpp:294] Iteration 50000, Testing net
(#0)
I0917 14:20:02.812610 19415 solver.cpp:343] Test net output #0: loss =
0.0183358 (* 1 = 0.0183358 loss)
I0917 14:20:02.812720 19415 solver.cpp:281] Optimization Done.
I0917 14:20:02.812728 19415 caffe.cpp:134] Optimization Done.

```

参考文献

- [1] Signature verification using a "Siamese" time delay neural network, Jane Bromley, Isabelle Guyon, Yann LeCun etc. 1993
- [2] Learning a similarity metric discriminatively, with application to face verification, Sumit Chopra, Raia Hadsell, Yann LeCun, 2005
- [3] Yangqing Jia, Evan Shelhamer, Caffe Tutorial, <http://caffe.berkeleyvision.org/tutorial/>, 2016

10

第 10 章 SqueezeNet 模型

CNN 网络模型一直在追求识别成功率，从 AlexNet 到 VGGNet 模型，识别精度不断提升，Top-5 错误率从 15.3% 下降到 7.3%，但参数量也越来越多，从 60M 增加到 140M。GoogLeNet 模型由于采用 Inception 结构，其参数量有明显的减少，只有 7M 大小。但这仍然难以在容量非常有限的 FPGA 硬件上运行，过多的参数降低了分布式训练的效率，也给数据传输所需的网络宽带造成很大的负担。如何在保证识别精度的情况下，对网络参数进行压缩是需要进一步改进的方向。

本章介绍的 SqueezeNet 模型（压缩模型）就为了解决这样一个问题而提出的网络模型。本章我们先阐述 SqueezeNet 模型的基本理论，然后解读其 Caffe 的实现，最后介绍 Caffe 环境下的训练方法。

10.1 SqueezeNet 网络模型

10.1.1 SqueezeNet 模型原理

2016 年 UC Berkeley 和 Stanford 研究人员在论文“SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”^[1]提出了 SqueezeNet 网络模型的结构和设

计思想。SqueezeNet 的设计目标不是为了得到最佳的 CNN 识别精度，而是希望在简化网络复杂度的同时，保证网络模型的识别精度。

SqueezeNet 模型的设计用以下三个方法简化网络复杂度：

(1) 替换 3×3 的卷积核为 1×1 的卷积核。从 AlexNet 模型发展到现在，因为设计上的简洁和有效性，卷积核的大小大都选择 3×3 。SqueezeNet 模型用 1×1 的卷积核替换 3×3 的卷积核，可以让网络参数缩小 9 倍。但是为了不影响识别精度，只做了部分替换。

(2) 减少输入 3×3 卷积的输入特征数量。将卷积层分解为 squeeze 层和 expand 层，并封装成为一个 Fire Module。

(3) 在整个网络后期进行下采样，使得卷积层有比较大的 activation maps。

10.1.2 Fire Module

Fire Module 是 SqueezeNet 模型的核心构件，其思想非常简单。即将一个卷积层分解成一个 squeeze 层和一个 expand 层，并各自带上 ReLU 激活层。squeeze 层包含全部是 1×1 的卷积核，共有 S 个。expand 层包含 1×1 和 3×3 的卷积核，其中 1×1 的卷积核有 $E1$ 个， 3×3 的卷积核有 $E3$ 个，要求满足 $S < (E1 + E3)$ 。如图 10-1 所示的 Fire Module 结构。

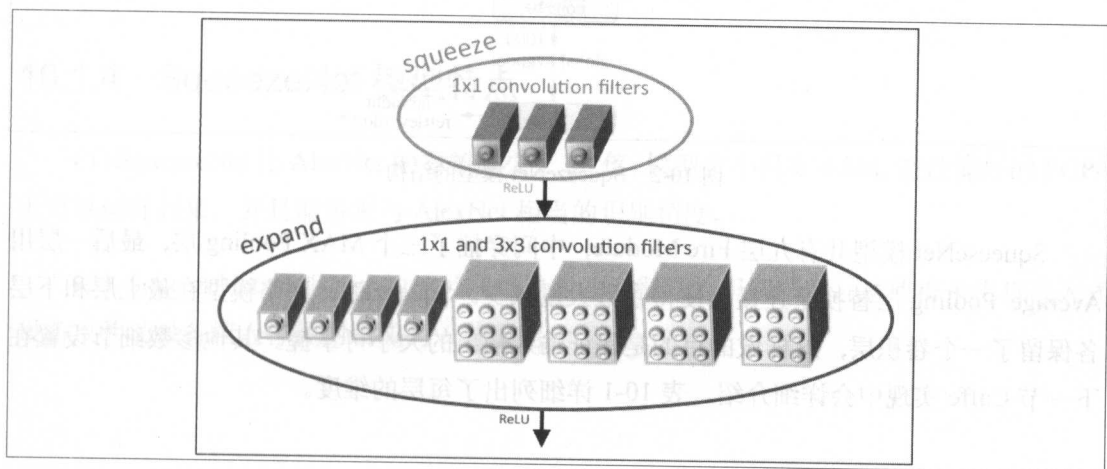


图 10-1 Fire Module 结构

10.1.3 SqueezeNet 模型结构

SqueezeNet 模型的结构如图 10-2 所示。

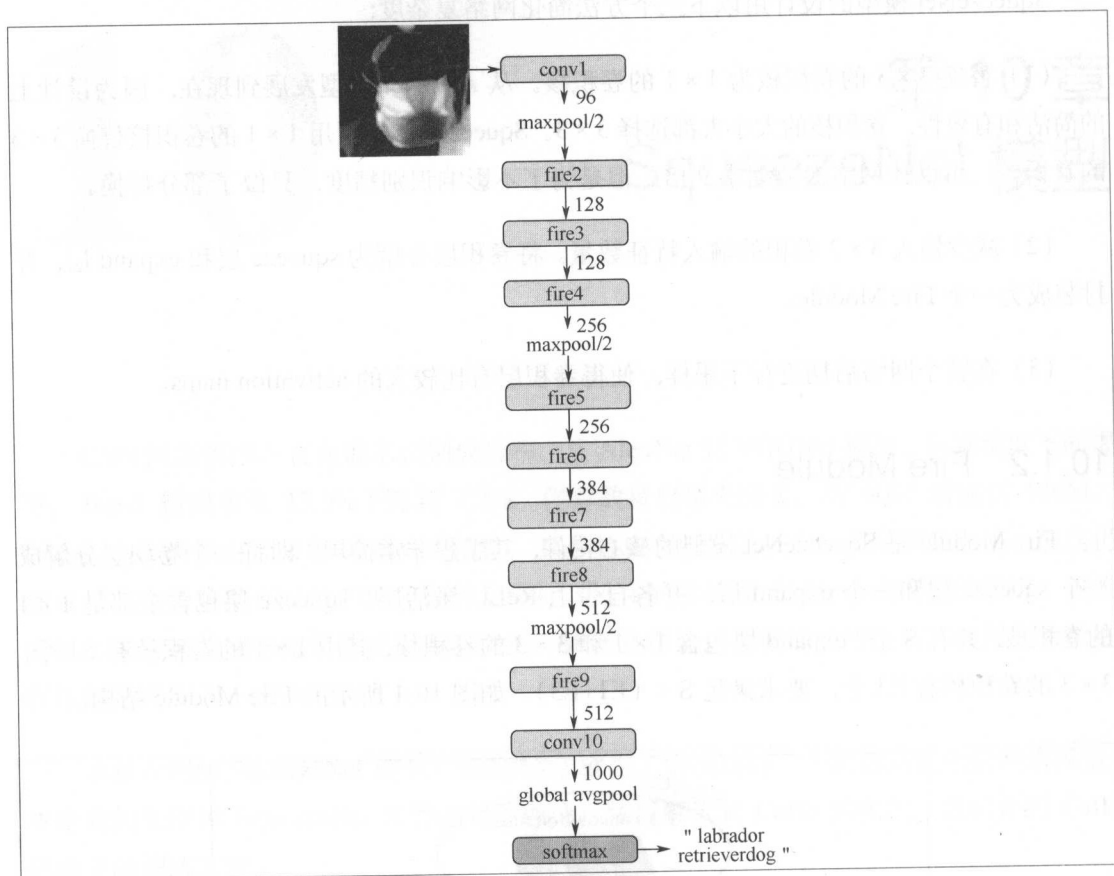


图 10-2 SqueezeNet 模型的结构

SqueezeNet 模型共有九层 Fire Module，中间穿插了三个 MAX Pooling 层，最后一层用 Average Pooling 层替换全连接层使得参数大量减少。SqueezeNet 网络模型在最上层和下层各保留了一个卷积层，这样做的目的是保证输入输出的大小可掌握。其他参数细节设置在下一节 Caffe 实现中会详细介绍。表 10-1 详细列出了每层的维度。

表 10-1 SqueezeNet 模型的网络维度

layer name/type	output size	filter size/ stride (if not a fire layer)	depth	$S_{1 \times 1}$ (#1×1 squeeze)	$e_{1 \times 1}$ (#1×1 expand)	$e_{3 \times 3}$ (#3×3 expand)
input image	224×224×3					
conv1	111×111×96	7×1/2(×96)	1			
maxpool1	55×55×96	3×3/2	0			
fire2	55×55×128		2	16	64	64
fire3	55×55×128		2	16	64	64
fire4	55×55×256		2	32	128	128
maxpool4	27×27×256	3×3/2	0			
fire5	27×27×256		2	32	128	128
fire6	27×27×384		2	48	192	192
fire7	27×27×384		2	48	192	192
fire8	27×27×512		2	64	256	256
maxpool8	13×12×512	3×3/2	0			
fire9	13×12×512		2	64	256	256
conv10	13×12×1000	1×1/1(×1000)	1			
avgpool10	1×1×1000	13×13×1	0			

activations

parameters

10.1.4 SqueezeNet 模型特点

(1) SqueezeNet 比 AlexNet 的参数减少了 50 倍, 模型大小只有 4.8M, 在性能好的 FPGA 上可以运行起来, 并且能带来与 AlexNet 相当的识别精度。

(2) SqueezeNet 证明了小的神经网络也能达到很好的识别精度, 这使得未来将嵌入式设备或移动设备植入神经网络成为一种可能。

10.2 SqueezeNet 网络实现

SqueezeNet 的网络实现在 github 上有下载, https://github.com/DeepScale/SqueezeNet/blob/master/SqueezeNet_v1.1/train_val.prototxt。具体内容和注释如下:

```
# please cite:
# @article{SqueezeNet,
#   Author = {Forrest N. Iandola and Matthew W. Moskewicz and Khalid Ashraf
and Song Han and William J. Dally and Kurt Keutzer},
#   Title = {SqueezeNet: AlexNet-level accuracy with 50x fewer parameters
and <$1MB model size},
#   Journal = {arXiv:1602.07360},
#   Year = {2016}
# }
layer {                                // data 层
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    crop_size: 227
    mean_value: 104
    mean_value: 117
    mean_value: 123
  }
  data_param {
    source: "examples/imagenet/ilsrvrc12_train_lmdb" //训练数据集
    batch_size: 32 #*iter_size
    backend: LMDB
  }
}
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    crop_size: 227
    mean_value: 104
    mean_value: 117
    mean_value: 123
  }
}
```



```

data_param {
  source: "examples/imagenet/ilsrvrc12_val_lmdb" // 测试数据集
  batch_size: 25 #not *iter_size
  backend: LMDB
}
}
layer { //第一个卷积层, 缩小输入图片, 提取 96 维特征
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 96
    kernel_size: 7
    stride: 2
    weight_filler {
      type: "xavier"
    }
  }
}
}
layer { //ReLU 层
  name: "relu_conv1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
}
layer { //第一个 MAX Pooling 层, 降采样, 缩小一半
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
}
layer { // 第一个 fire 模块, 模块内部先用 squeeze 层减少通道数, 再用 expand 层
  name: "fire2/squeeze1x1"
  type: "Convolution"
  bottom: "pool1"
  top: "fire2/squeeze1x1"
  convolution_param {
    num_output: 16
    kernel_size: 1
    weight_filler {
      type: "xavier"
    }
  }
}
}
}

```

增加通道数


```

layer {
  name: "fire2/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire2/squeeze1x1"
  top: "fire2/squeeze1x1"
}
layer {
  name: "fire2/expand1x1"
  type: "Convolution"
  bottom: "fire2/squeeze1x1"
  top: "fire2/expand1x1"
  convolution_param {
    num_output: 64
    kernel_size: 1
    weight_filler {
      type: "xavier"
    }
  }
}
layer {
  name: "fire2/relu_expand1x1"
  type: "ReLU"
  bottom: "fire2/expand1x1"
  top: "fire2/expand1x1"
}
layer {
  name: "fire2/expand3x3"
  type: "Convolution"
  bottom: "fire2/squeeze1x1"
  top: "fire2/expand3x3"
  convolution_param {
    num_output: 64
    pad: 1      // 增加一个边界像素, 使得 1*1 和 3*3 filter 对齐。
    kernel_size: 3
    weight_filler {
      type: "xavier"
    }
  }
}
layer {
  name: "fire2/relu_expand3x3"
  type: "ReLU"
  bottom: "fire2/expand3x3"
  top: "fire2/expand3x3"
}
layer {
  name: "fire2/concat"
  type: "Concat"
  bottom: "fire2/expand1x1"
  bottom: "fire2/expand3x3"
  top: "fire2/concat"
}

```

```

}
... // 中间共 9 个 fire 模块作用相同，这里省略。
layer {                                // 最后一个 fire 模块后，增加一个 Dropout 层
    name: "drop9"
    type: "Dropout"
    bottom: "fire9/concat"
    top: "fire9/concat"
    dropout_param {
        dropout_ratio: 0.5 // 丢弃率为 50%
    }
}

layer {                                // 第二个卷积层，为图的每个像素预测 1000 个分类
    name: "conv10"
    type: "Convolution"
    bottom: "fire9/concat"
    top: "conv10"
    convolution_param {
        num_output: 1000
        pad: 1
        kernel_size: 1
        weight_filler {
            type: "gaussian"
            mean: 0.0
            std: 0.01
        }
    }
}

layer {
    name: "relu_conv10"
    type: "ReLU"
    bottom: "conv10"
    top: "conv10"
}

layer {                                // average pooling 层，得到 1000 类。
    name: "pool10"
    type: "Pooling"
    bottom: "conv10"
    top: "pool10"
    pooling_param {
        pool: AVE
        global_pooling: true
    }
}

layer {                                // softmax 层，使用 softmax 函数归一化为概率。
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "pool10"
    bottom: "label"
    top: "loss"
    include {
        phase: TRAIN
    }
}

```

```
    }  
  }  
  layer {  
    name: "accuracy"  
    type: "Accuracy"  
    bottom: "pool10"  
    bottom: "label"  
    top: "accuracy"  
    include {  
      phase: TEST  
    }  
  }  
  layer {  
    name: "accuracy_top5"  
    type: "Accuracy"  
    bottom: "pool10"  
    bottom: "label"  
    top: "accuracy_top5"  
    include {  
      phase: TEST  
    }  
    accuracy_param {  
      top_k: 5  
    }  
  }  
}
```

11

第 11 章 FCN 模型

在本书前面的章节中，我们可以看到卷积网是非常强大的，在物体分类的任务中起着非常重要的作用。本章介绍的是全卷积网络（Full Convolutional Network），首先介绍 FCN 网络模型的特点，然后对 FCN 网络进行具体的解读。

11.1 FCN 模型简介

FCN（Full Convolutional Network）的概念初始提出是为了解决语义级别图像分割（semantic image segmentation）的问题，使得图像对应位置的每个像素都可进行分类。但 FCN 的应用不仅仅于此，自从 FCN 提出后，一些经典的物体检测和识别模型都采用了 FCN 的概念，比如最新的 Faster-RCNN 模型和 SSD 模型。

FCN 与经典的 CNN 在卷积层之后使用全连接层获得固定长度的特征向量进行分类所不同，FCN 作者敏锐地意识到了在经典的卷积网中输入图像尺寸必须固定的这一限制来自于网络最后的全连接网，由于全连接网必须是固定尺寸输入，导致整个网络的输入尺寸无法改变。但 CNN 的特性是可以接受任意尺寸的输入图像，这就产生了一个想法，我们是否可以用 FCN 来替代全连接网？论文“Fully Convolutional Networks for Semantic Segmentation”^[1]中采用反卷积层对最后一个卷积层的 feature map 进行上采样，使它恢复到

输入图像相同的尺寸，从而可以对每个像素都产生了一个预测，同时也保留了原始输入图像中的空间信息，最后在上采样的特征图上进行逐个像素分类。

从图 11-1 中，可以清楚地看到全连接网络和 FCN 之间的关系。

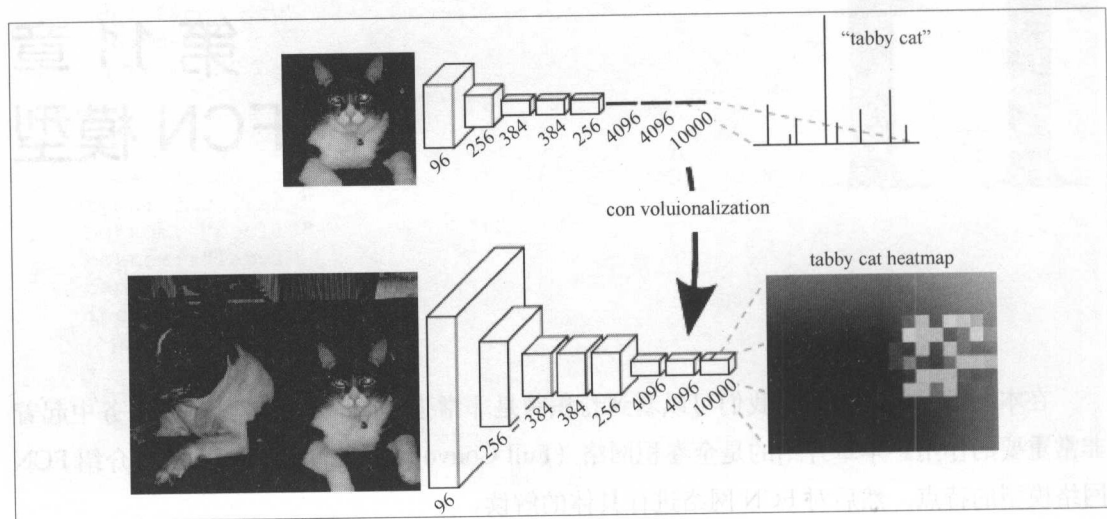


图 11-1 全连接网络和 FCN 之间的关系

11.2 FCN 的特点和使用场景

与传统用 CNN 进行图像分割的方法相比，FCN 有两大明显的优点：一是可以接受任意大小的输入图像，而不再要求所有的训练图像和测试图像具有同样的尺寸；二是更加高效，因为避免了由于使用像素块而带来的重复存储和计算卷积的问题。

同时 FCN 的缺点也比较明显：一是得到的结果还是不够精细。进行 8 倍上采样虽然比 32 倍的效果好了很多，但是上采样的结果还是比较模糊和平滑，对图像中的细节不敏感。二是对各个像素进行分类，没有充分考虑像素与像素之间的关系，忽略了在通常的基于像素分类的分割方法中使用的空间规整（spatial regularization）步骤，缺乏空间一致性。

FCN 这种全新的思路开辟了一个新的图像分割方向，对计算机视觉领域的影响是十分巨大的。当前已有大量的算法基于 FCN。

FCN 能够端到端 (end to end) 地得到每个像素的预测结果，目前也涌现了一大批基于 FCN 的算法，例如边缘检测 (edge detection)、视觉跟踪 (visual tracking) 等。同时 FCN 也可以省去传统识别中复杂的逐个 patch 计算过程。

11.3 Caffe FCN 解读

FCN 基于 Caffe 框架的全部实现在 Github 上是开源的。我们可以通过 git 工具直接下载。在 Linux 下输入：

```
$ git clone https://github.com/BVLC/caffe/fcn.berkeleyvision.org.git
```

执行上述命令后，在当前目录下可看到 fcn.berkeleyvision.org 这个目录。由于不同的数据源和不同的 FCN 类型的网络结构并不相同，对数据源的读取方式也各不相同，因此可以看到大量的分支，每个目录都是一个分支，具体分支如下：

- nyud-fcn32s-color
- nyud-fcn32s-color-d
- nyud-fcn32s-color-hha
- nyud-fcn32s-hha
- pascalcontext-fcn16s
- pascalcontext-fcn32s
- pascalcontext-fcn8s
- siftflow-fcn16s

- siftflow-fcn32s
- siftflow-fcn8s
- voc-fcn16s
- voc-fcn32s
- voc-fcn8s
- voc-fcn8s-atonce
- voc-fcn-alexnet

本书选择其中一个代表性的分支 `pascalcontext-fcn32s` 作为研究对象。

11.3.1 FCN 模型训练准备

在训练 FCN 的 `fcncontext-fcn32s` 分支之前, 请先确保 Caffe 已正确安装, 且 `make` 和 `make pycaffe` 成功, 另外所有的路径设置正确。

1. 下载VOC2010数据集

请读者自行搜索安装 VOC2010 数据集, 原始数据集至少包括以下两个文件:

- VOCdevkit_08-May-2010.tar
- VOCTrainval_03-May-2010.tar

用 `tar -xvf` 解压缩后, 生成 VOCdevkit 目录, 内部有 VOC2010 子目录, 然后输入:

```
$ ln -s VOCdevkit/VOC2010 fcn.berkeleyvision.org/data/pascal/VOC2010
```

2. 下载pascal-context数据集

下载 pascal-context 数据集的步骤如下:

- (1) 访问 <http://www.cs.stanford.edu/~roozbeh/pascal-context>;
- (2) 下载 59_context_labels.tar 并解压到 fcn.berkeleyvision.org/data/pascal-context 下;
- (3) 下载 trainval.tar 并解压到 fcn.berkeleyvision.org/data/pascal-context 下。

3. 下载VGG16预训练caffemodel

下载 VGG16 预训练 caffemodel 的步骤如下:

- (1) 访问

<https://github.com/BVLC/caffe/wiki/Model-Zoo#models-used-by-the-vgg-team-in-ilsvrc-2014>;

- (2) 下载 VGG_ILSVRC_16_layers.cafemodel^[2]到 fcn.berkeleyvision.org 下, 并改名为 vgg16fc.cafemodel。

注: 由于国内网络限制的缘故, 可能导致无法访问, 请读者自行解决。

4. Label文件重命名

```
$ ln -s classes-59.txt 59_label.txt
$ ln -s classes-400.txt labels.txt
```

添加一行到 labels.txt 的第一行, 内容为 0: background。

5. 添加python目录

如果 fcn.berkeleyvision.org 不在 python 搜索的目录里时, 则编辑 ~/.bashrc, 增加以下行到 .bashrc 中。

```
export PYTHONPATH=yourpath/fcn.berkeleyvision.org:$PYTHONPATH
```

然后 logout 后重新登录使其生效。

6. 更改相关路径

更改 `pascalcontext-fcn32s/train.prototxt` 和 `val.prototxt` 里的路径, 把 `param_str` 中的 `context_dir` 和 `voc_dir` 更改为自己的路径。

比如, 只需 `../data/pascal-context` 更改为 `../data/pascal-context`, 把 `../data/pascal` 更改为 `../data/pascal`。

7. 手动创建目录

在 `fcn.berkeleyvision.org/pascalcontext-fcn32s` 下创建 `snapshot/train`。

8. 更改层名

由于下载的 `vgg16fc.caffemodel` 中也有 `fc6`, `fc7`, 和 `train.txt`, `val.txt` 中 `fc6`, `fc7` 不一致, 会导致错误。所以把 `train.txt`, `val.txt` 中的所有 `fc6` 更名为 `fc6x`, 所有 `fc7` 更名为 `fc7x`, 包括层名和 blob 名。

9. 更正 solve.py 的 Bug

请在 `solve.py` 的最前面增加 `import sys`。

10. 安装 python setproctitle 包

如果系统缺少 `setproctitle` 包, 请执行以下命令安装:

```
$ pip install setproctitle
```

至此, 所有准备工作做完, 整个过程相对复杂, 但情有可原, 因为在 FCN 的下载页面上, 原作者提到过项目还未完工。

11.3.1 FCN 模型训练

1. 训练过程

在 pascalcontext-fcn32s 下, 输入以下命令开始训练:

```
$python solve.py 0
```

注: 0 代表 GPU 0, 读者如果有多 GPU 的话, 应根据电脑的 GPU 情况修改。

训练需要相当长的时间, 如果读者不想亲自训练的话, 也可以在网上直接下载已训练好的 caffemodel, 下载的 URL 在 pascalcontext-fcn32s 下的 caffemodel-url 中。

2. solver.prototxt解读

```
train_net: "train.prototxt" #训练用的网络 prototxt 文件
test_net: "val.prototxt" #测试用的网络 prototxt 文件
test_iter: 5105 #测试时的迭代次数
# make test net, but don't invoke it from the solver itself
test_interval: 99999999 #solver 自身不会发起测试
display: 20 #每 20 次迭代显示一次迭代信息
average_loss: 20 #每 20 次迭代平均一下 loss, 以防止 loss 的跳变
lr_policy: "fixed" #固定类型的学习率
# lr for unnormalized softmax
base_lr: 1e-10 #基本学习率
# high momentum
momentum: 0.99 #高 momentum
# no gradient accumulation
iter_size: 1
max_iter: 30000 #最大迭代次数
weight_decay: 0.0005
snapshot: 4000 #每 4000 次迭代抓一次快照
snapshot_prefix: "snapshot/train" #快照的路径
test_initialization: false #不做测试初始化
```

3. train.prototxt解读

```
layer {
  name: "data"
  type: "Python" #Python 类型的层, 要使其生效, 必须在 Caffe 的 Makefile.config 中
  设置 WITH_PYTHON_LAYER := 1
  top: "data" #top blob 0, 为图像数据
```

```

top: "label" #top blob 1, 为图像数据的标签
python_param {
    module: "pascalcontext_layers" #python 模块名, 对应 pascalcontext_layers.py
    layer: "PASCALContextSegDataLayer" #类名, FCN 的原作者用自定义的 Python 层来
处理原始数据
    param_str: "{\'context_dir\': \'../data/pascal-context\', \'seed\':
1337, \'split\': \'train\', \'voc_dir\': \'../data/pascal/\'}"
    } #Python 层所用参数
}
layer {
    name: "conv1_1" #卷积层 conv1_1
    type: "Convolution"
    bottom: "data" #输入为 blob "data"
    top: "conv1_1"
    param {
        lr_mult: 1 #w 学习率乘数为 1
        decay_mult: 1
    }
    param {
        lr_mult: 2 #偏置 b 学习率乘数为 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 64 #输出为 64 channels
        pad: 100 #边缘填充, 每边 100 像素
        kernel_size: 3 #3x3 尺寸的卷积核
        stride: 1
    }
}
layer {
    name: "relu1_1" #ReLU 层 relu1_1
    type: "ReLU"
    bottom: "conv1_1" #输入的 blob 经过 ReLU 处理后, 数据还是放在原有 blob 中
    top: "conv1_1"
}
layer {
    name: "conv1_2" #卷积层 conv1_2
    type: "Convolution"
    bottom: "conv1_1"
    top: "conv1_2"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {

```



```

    num_output: 64 #输出为 64 channels
    pad: 1
    kernel_size: 3
    stride: 1
  }
}
layer {
  name: "relu1_2" #ReLU层 relu1_2
  type: "ReLU"
  bottom: "conv1_2"
  top: "conv1_2"
}
layer {
  name: "pool1" #Pooling层 pool1
  type: "Pooling"
  bottom: "conv1_2"
  top: "pool1"
  pooling_param {
    pool: MAX #MAX 类型的 Pooling, 取 2x2 中的最大值
    kernel_size: 2
    stride: 2 #步长为 2
  }
}
layer {
  name: "conv2_1" #卷积层 conv2_1
  type: "Convolution"
  bottom: "pool1"
  top: "conv2_1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 128
    pad: 1
    kernel_size: 3
    stride: 1
  }
}
layer {
  name: "relu2_1" #ReLU层 relu2_1
  type: "ReLU"
  bottom: "conv2_1"
  top: "conv2_1"
}
layer {

```



```

name: "conv2_2" #卷积层 conv2_2
type: "Convolution"
bottom: "conv2_1"
top: "conv2_2"
param {
  lr_mult: 1
  decay_mult: 1
}
param {
  lr_mult: 2
  decay_mult: 0
}
convolution_param {
  num_output: 128
  pad: 1
  kernel_size: 3
  stride: 1
}
}
layer {
  name: "relu2_2" #ReLU层 relu2_2
  type: "ReLU"
  bottom: "conv2_2"
  top: "conv2_2"
}
layer {
  name: "pool2" #Pooling层 pool2
  type: "Pooling"
  bottom: "conv2_2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv3_1" #卷积层 conv3_1
  type: "Convolution"
  bottom: "pool2"
  top: "conv3_1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256

```

```

        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu3_1" #ReLU层 relu3_1
    type: "ReLU"
    bottom: "conv3_1"
    top: "conv3_1"
}
layer {
    name: "conv3_2" #卷积层 conv3_2
    type: "Convolution"
    bottom: "conv3_1"
    top: "conv3_2"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256
        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu3_2" #ReLU层 relu3_2
    type: "ReLU"
    bottom: "conv3_2"
    top: "conv3_2"
}
layer {
    name: "conv3_3" #卷积层 conv3_3
    type: "Convolution"
    bottom: "conv3_2"
    top: "conv3_3"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {

```

```

        num_output: 256
        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu3_3" #ReLU层 relu3_3
    type: "ReLU"
    bottom: "conv3_3"
    top: "conv3_3"
}
layer {
    name: "pool3" #Pooling层 pool3
    type: "Pooling"
    bottom: "conv3_3"
    top: "pool3"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
layer {
    name: "conv4_1" #卷积层 conv4_1
    type: "Convolution"
    bottom: "pool3"
    top: "conv4_1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 512
        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu4_1" #ReLU层 relu4_1
    type: "ReLU"
    bottom: "conv4_1"
    top: "conv4_1"
}
layer {
    name: "conv4_2" #卷积层 conv4_2

```



```

    type: "Convolution"
    bottom: "conv4_1"
    top: "conv4_2"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 512
        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu4_2" #ReLU层 relu4_2
    type: "ReLU"
    bottom: "conv4_2"
    top: "conv4_2"
}
layer {
    name: "conv4_3" #卷积层 conv4_3
    type: "Convolution"
    bottom: "conv4_2"
    top: "conv4_3"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 512
        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu4_3" #ReLU层 relu4_3
    type: "ReLU"
    bottom: "conv4_3"
    top: "conv4_3"
}
layer {

```

```

name: "pool4" #Pooling 层 pool4
type: "Pooling"
bottom: "conv4_3"
top: "pool4"
pooling_param {
  pool: MAX
  kernel_size: 2
  stride: 2
}
}
layer {
  name: "conv5_1" #卷积层 conv5_1
  type: "Convolution"
  bottom: "pool4"
  top: "conv5_1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 512
    pad: 1
    kernel_size: 3
    stride: 1
  }
}
layer {
  name: "relu5_1" #ReLU 层 relu5_1
  type: "ReLU"
  bottom: "conv5_1"
  top: "conv5_1"
}
layer {
  name: "conv5_2" #卷积层 conv5_2
  type: "Convolution"
  bottom: "conv5_1"
  top: "conv5_2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 512

```

```

        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu5_2" #ReLU层 relu5_2
    type: "ReLU"
    bottom: "conv5_2"
    top: "conv5_2"
}
layer {
    name: "conv5_3" #卷积层 conv5_3
    type: "Convolution"
    bottom: "conv5_2"
    top: "conv5_3"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 512
        pad: 1
        kernel_size: 3
        stride: 1
    }
}
layer {
    name: "relu5_3" #ReLU层 relu5_3
    type: "ReLU"
    bottom: "conv5_3"
    top: "conv5_3"
}
layer {
    name: "pool5" #Pooling层 5, 至此, VGG 的所有卷积部分结束, 下面应为 VGG 的全连接层,
    type: "Pooling"
    bottom: "conv5_3"
    top: "pool5"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
layer {
    name: "fc6" #卷积层 fc6

```

在 FCN 中替代为卷积层, 起全连接作用


```

type: "Convolution"
bottom: "pool5"
top: "fc6"
param {
  lr_mult: 1
  decay_mult: 1
}
param {
  lr_mult: 2
  decay_mult: 0
}
convolution_param {
  num_output: 4096 #4096 channels, 相当于原有第一个全连接层的 4096 个神经元
  pad: 0
  kernel_size: 7 #7x7 卷积核, 对 Pooling 层 5 的 7x7 区域做识别
  stride: 1
}
}
layer {
  name: "relu6" #ReLU 层 relu6
  type: "ReLU"
  bottom: "fc6"
  top: "fc6"
}
layer {
  name: "drop6" #Dropout 层 drop6
  type: "Dropout"
  bottom: "fc6"
  top: "fc6"
  dropout_param {
    dropout_ratio: 0.5 #drop 概率为 0.5
  }
}
layer {
  name: "fc7" #卷积层 fc7
  type: "Convolution"
  bottom: "fc6"
  top: "fc7"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 4096 #4096 channels, 相当于原有第二个全连接层的 4096 个神经元
    pad: 0
    kernel_size: 1 #1 符合原有两个全连接层之间的相连, 即第二个 FC 层的 4096 channels
  }
}

```

上的点和第一个 FC 层的 4096 channels 上相应位置的点完全连接

```

    stride: 1
  }
}
layer {
  name: "relu7" #ReLU 层 relu7
  type: "ReLU"
  bottom: "fc7"
  top: "fc7"
}
layer {
  name: "drop7" #Dropout 层 drop7
  type: "Dropout"
  bottom: "fc7"
  top: "fc7"
  dropout_param {
    dropout_ratio: 0.5
  }
}
layer {
  name: "score_fr" #卷积层 score_fr, 相当于又一个全连接层, 但输出分类的 score
  type: "Convolution"
  bottom: "fc7"
  top: "score_fr"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 60 #60 代表 60 类图像, 这里 score_fr 层上的每个点都有 60 个分类中不同
    pad: 0
    kernel_size: 1
  }
}
layer {
  name: "upscore" #上卷积层, 用于图像分割
  type: "Deconvolution"
  bottom: "score_fr"
  top: "upscore"
  param {
    lr_mult: 0
  }
  convolution_param {
    num_output: 60
    bias_term: false
  }
}

```

```

        kernel_size: 64
        stride: 32
    }
}
layer {
    name: "score" #Crop 层, 主要是全卷积时原始图像加了 pad, 比原图大一些, 最后要把 pad
裁剪掉
    type: "Crop"
    bottom: "upscore" #bottom[0]中为要被 crop 的层
    bottom: "data" #bottom[1]中为按这个层的尺寸 crop bottom[0], 简而言之, 就是把
upscore 层 crop 到 data 层的尺寸
    top: "score"
    crop_param {
        axis: 2 #从 axis 2 开始 crop, 其实就是 crop axis 2 和 axis 3, 即 width 和 height
        offset: 19 #crop 的偏移量, upscore 层的尺寸比 data 层的尺寸大, offset 指示从那
个位置 crop, 只有一个 offset, 意思是 height 和 weight 用同一个 offset
    }
}
layer {
    name: "loss" #SoftmaxWithLoss 层, 用于训练
    type: "SoftmaxWithLoss"
    bottom: "score"
    bottom: "label"
    top: "loss"
    loss_param {
        ignore_label: 255
        normalize: false
    }
}
}

```

val.prototxt 和 train.prototxt 几乎一样, 除了 data 层用的是 PASCALContextSegDataLayer, 其他没有区别, 这里不再赘述。

4. 训练日志摘要

```

I0722 11:35:04.433732 6743 solver.cpp:48] Initializing solver from
parameters:
train_net: "train.prototxt"
test_net: "val.prototxt"
test_iter: 5105
test_interval: 999999999
base_lr: 1e-10
display: 20
max_iter: 300000
lr_policy: "fixed"
momentum: 0.99
weight_decay: 0.0005

```

```
snapshot: 4000
snapshot_prefix: "snapshot/train"
test_initialization: false
average_loss: 20
iter_size: 1
I0722 11:35:04.433794 6743 solver.cpp:81] Creating training net from
train_net file: train.prototxt
...
...
```

参考文献

- [1] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In CVPR, 2015.
- [2] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.

12

第 12 章 R-CNN 模型

在本书前面章节中，我们介绍了经典的物体分类模型，但在计算机视觉的实际应用中，分类是远远不够的，更多的场景需要能同时探测到多个物体，不仅包括这些物体的分类，而且包括这些物体的位置和尺寸。在计算机视觉中，我们通常用 Bounding Box（把探测物体的形状及其子形状紧密封闭在内的矩形）来表示这些物体的位置和尺寸。本章所介绍的 R-CNN^[1]模型就是一个经典的目标检测模型。

12.1 R-CNN 模型简介

R-CNN 模型发展到现在已经成为一个系列，从最初的 R-CNN 模型发展到 Fast-RCNN^[2]模型，之后在 2015 年再一次突破，提出了 Faster-RCNN^[3]模型。近几年来，基于深度学习的目标检测方法发展很快，提出了很多精妙的思想，R-CNN 模型系列功不可没。R-CNN、Fast-RCNN、Faster-RCNN 代表了目标检测的前沿水平，在 Github 上都给出了基于 Caffe 框架的源码。

R-CNN 模型由于提出较早，相比今天很多前沿方法，它的精度和效率都已经略显不足，但是该模型是很多算法的基础思想，对 R-CNN 模型做深入研究是学习深度学习目标检测模型的必然步骤。

R-CNN 结合了 Selective Search^[4]算法的区域建议框和由卷积神经网络产生的视觉丰富特征,这在当时是一个创新。在 R-CNN 发布的时候,R-CNN 比当时最好的检测模型在 Pascal VOC 2012 数据集上有差不多 30%的性能提升。mAP (mean average precision) 从 40.9%提升到 53.3%。和以前最好的结果不同,基于深度学习算法的 R-CNN 获取这一成绩完全没有采用人工提取视觉特征的方法。

R-CNN 最开始在一份 arXiv tech report (<http://arxiv.org/abs/1311.2524>) 公布,其后论文被 CVPR 2014 接受。有兴趣的读者可参阅论文“Rich feature hierarchies for accurate object detection and semantic segmentation”。

12.2 R-CNN 的特点和使用场景

R-CNN 是深度学习在目标检测任务上的应用尝试,其中 R 对应于“Region(区域)”,整体框架与传统方法相似。

R-CNN 有两个关键点:一是使用建议窗口,并用 CNN 对其进行特征提取;二是样本缺乏时,使用大量辅助样本预先训练,再用自己的样本进行微调。

R-CNN 的整体框架大致为:

- 采用 Selective Search 生成建议窗口。R-CNN 用 Selective Search 方法生成建议窗口 (proposals), 每张图片大约生成 2000 个建议窗口, 由于建议窗口尺寸不一, 所以通过拉伸 (warp), 形成统一的输入尺寸 227×227 。
- 运用 CNN 进行特征提取。把统一尺寸的建议窗图像输入 CNN, 提取 CNN 特征。
- 对建议窗图像进行分类 (SVM), 用 SVM 对 CNN 输出特征分类。
- Bounding box 回归 (Bounding box regression), 用 Bounding box 回归值校正原来的建议窗口, 生成预测窗口坐标。

由于用于目标检测的数据集较小, 无法充分从零开始训练网络, R-CNN 采用在预训练

网络上微调的方法。即通过 ILSVRC 2012 数据集对 CNN 进行预训练, 然后用 PASCL VOC 数据集再进行微调训练。R-CNN 模型结构图如图 12-1 所示。

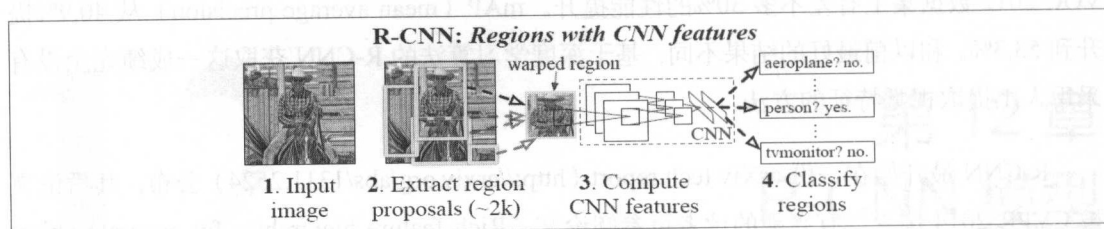


图 12-1 R-CNN 模型结构图

R-CNN 在各数据集上的表现如表 12-1 所示。

表 12-1 R-CNN 在各数据集上的表现

模型	VOC2007 mAP	VOC2010 mAP	VOC2012 mAP	ILSVRC2013 test mAP
R-CNN	54.2%	50.2%	49.6%	N/A
R-CNN bbox reg	58.5%	53.7%	53.3%	31.4%

R-CNN 相比于当前的前沿算法, 精确度不够高, 速度不够快, 无法满足实时系统需要, 但 R-CNN 第一次让业界看到了深度学习的力量, 以及在目标检测上的潜力, 是深度学习在目标检测任务上迈出的重要一步。

12.3 Caffe R-CNN 解读

R-CNN 基于 Caffe 框架的全部实现在 Github 上是开源的。可以按以下步骤来 DIY 训练 R-CNN。

12.3.1 R-CNN 模型训练准备

1. 环境准备

由于 R-CNN 主要的实现是基于 MATLAB 的, R-CNN 代码也基于 MATLAB 2012b on

64-bit Linux 版本上完整测试过，因此必须安装 MATLAB。

安装 Caffe。请读者注意，R-CNN 不兼容当前最新的 Caffe Master 版本，必须安装 Caffe V0.999 版本来保证兼容性。

Caffe V0.999 版本的获取和安装步骤如下：

(1) 下载 Caffe v0.999 (<https://github.com/BVLC/caffe/archive/v0.999.tar.gz>)。

(2) 正常安装 Caffe V0.999。

(3) 在 make 完 Caffe 后，执行 make matcaffe 安装 Caffe MATLAB Interface。

2. 下载R-CNN源代码并安装

可通过 git 工具直接下载 R-CNN 源代码，在 Linux 命令行环境的具体方法如下：

```
$ git clone https://github.com/rbgirshick/rcnn.git
$ cd rcnn
$ ln -sf $CAFFE_ROOT external/caffe
```

其中\$CAFFE_ROOT 为用户本机上 caffe V0.999 的安装目录。

3. 编译R-CNN

在 R-CNN 目录下，运行 MATLAB。

在出现 MATLAB 提示符“>>”后，执行 rcnn_build()。rcnn_build()会编译 liblinear 和 Selective Search。如果在编译过程中出现 warning 信息，这是正常的。

(1) 检查 Caffe 和 MATLAB Interface 是否正常安装

在 MATLAB 提示符“>>”下运行 key = caffe('get_init_key')，如果输出 key = -2，则表示安装正常。

注意：在启动 MATLAB 前，必须设置 LD_LIBRARY_PATH。如果输出如下的打印信

息 “Invalid MEX-file '/path/to/rcnn/external/caffe/matlab/caffe/caffe.mexa64': libmkl_rt.so: cannot open shared object file: No such file or directory”, 请确保 CUDA and MKL 在 LD_LIBRARY_PATH 中。LD_LIBRARY_PATH 设置方法如下 (如果使用的是 Intel MKL 库):

编辑 bash.bashrc:

```
$export LD_LIBRARY_PATH=/opt/intel/mkl/lib/intel64:/usr/local/cuda/lib64
```

(2) 下载已训练模型

运行 R-CNN 模型的最快方式是下载已训练的 R-CNN 模型数据。R-CNN 的原作者提供了多个已训练的 R-CNN 模型数据, 包括基于 PASCAL VOC 2007 train+val 数据集、PASCAL VOC 2012 train 数据集和 ILSVRC13 train+val 数据集的模型数据。整个已训练模型数据需要占用 1.5GB 硬盘空间。

下载步骤如下:

```
$ sh ./data/fetch_models.sh
$ sh ./data/fetch_selective_search_data.sh
```

(3) 运行已训练模型

在 R-CNN 安装目录下执行命令:

```
$ matlab
```

注意: 如果执行“matlab”后, 没有 “R-CNN startup done”字样打印输出, 则说明 MATLAB 没有在 R-CNN 安装目录下。

在 MATLAB 提示符下运行 RCNNdemo:

```
>> rcnn_demo
```

如果执行成功, 应该能看到一张图片, 上面有探测到的自行车和人, 如图 12-2 所示。下面所列的是探测到的物体的数值输出:

```
Top detection:
```

```
name
person      1.839884
swimming trunks -1.157806
turtle      -1.168884
tie         -1.217268
rubber eraser -1.246662
dtype: float32
```

Second-best detection:

```
name
bicycle      0.855625
unicycle    -0.334367
scorpion    -0.824552
lobster     -0.965544
lamp        -1.076225
dtype: float32
```

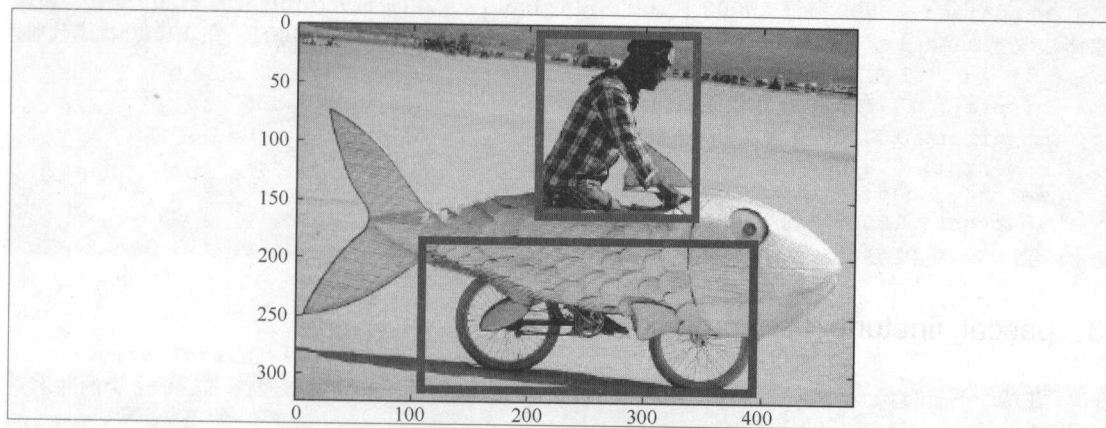


图 12-2 R-CNN 运行 Demo 演示图

12.3.2 R-CNN 模型训练

1. 训练过程

如果读者对训练感兴趣，可按下列步骤 DIY 训练。

(1) 提取特征到硬盘上；

(2) 训练 SVM；

(3) 测试。

具体的训练步骤这里不再展开, 请参考 R-CNN 目录下的 README.md。

2. pascal_finetune_solver.prototxt解读

```
train_net: "pascal_finetune_train.prototxt" #训练用的网络 prototxt 文件
test_net: "pascal_finetune_val.prototxt" #测试用的网络 prototxt 文件
test_iter: 100 #测试时的迭代次数
test_interval: 1000 #每 1000 次迭代测试一次
base_lr: 0.001 #基本学习率
lr_policy: "step" #step 类型的 lr 策略, 具体的 step 由 stepsize 决定
gamma: 0.1
stepsize: 20000 #每 20000 次迭代时改变 base_lr, 由于最大迭代次数为 100000 次, 所以
#会多次改变 base_lr, 即在 20000 次迭代时, base_lr 从 0.001 变为 0.0001, 在 40000 次迭代时,
base_lr 从 0.001 变为 0.00001, 以此类推
display: 20 #每 20 次迭代显示一次迭代信息
max_iter: 100000 #最大迭代次数
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000 #每 10000 次迭代抓一次快照
snapshot_prefix: "finetune_voc_2007_trainval"
```

3. pascal_finetune_train.prototxt解读

注意: 下面的 prototxt 文法只适用于 Caffe V0.999 版本, 和当前 caffe master 中的并不完全一致

```
name: "CaffeNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: WINDOW_DATA #WINDOW_DATA 类型的数据层
  window_data_param {
    source: "window_file_voc_2007_trainval.txt"
    mean_file: "../data/ilsrvcl2/imagenet_mean.binaryproto" #ILSVRC12
    #数据集的均值文件
    batch_size: 128 #batch 内有 128 张图片
    crop_size: 227 #crop 的尺寸是 227x227
    mirror: true #使用镜像图像
    fg_threshold: 0.5 #前景的门限设置, 和 ground truth 的 overlap 大于此门限判断为
    #识别物体 (positive 样本)
    bg_threshold: 0.5 #背景的门限设置, 和 ground truth 的 overlap 小于此门限判断为
    #背景 (negative 样本)
```



```

    fg_fraction: 0.25 #一个batch中 positive 样本数量的比例
    context_pad: 16 #clip 出的图像每边加 16 像素
    crop_mode: "warp" #crop 的方式为拉伸
  }
}
layers { #conv1 卷积层
  bottom: "data"
  top: "conv1"
  name: "conv1"
  type: CONVOLUTION
  blobs_lr: 1 #weight 的 lr 乘数
  blobs_lr: 2 #bias 的 lr 乘数
  weight_decay: 1
  weight_decay: 0
  convolution_param {
    num_output: 96 #输出 96 channel
    kernel_size: 11 #11x11 卷积核
    stride: 4 #卷积步长为 4
    weight_filler {
      type: "gaussian" #weight 高斯分布初始化
      std: 0.01
    }
    bias_filler {
      type: "constant" #bias 初始化为常数 0
      value: 0
    }
  }
}
}
layers { #relu1 为 ReLU 层
  bottom: "conv1"
  top: "conv1"
  name: "relu1"
  type: RELU
}
layers { #pool1 为 MAX pooling 层
  bottom: "conv1"
  top: "pool1"
  name: "pool1"
  type: POOLING
  pooling_param {
    pool: MAX
    kernel_size: 3 #3x3 pooling
    stride: 2 #步长为 2
  }
}
}
layers { #norm1 为 LRN 层
  bottom: "pool1"
  top: "norm1"
  name: "norm1"

```



```

type: LRN
lrn_param {
  local_size: 5 #5 个连续特征点做 LRN
  alpha: 0.0001 #LRN 公式中的 alpha 参数
  beta: 0.75 #LRN 公式中的 beta 参数
}
}

```

```

layers { #conv2 为卷积层
  bottom: "norm1"
  top: "conv2"
  name: "conv2"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  convolution_param {
    num_output: 256 #输出为 256 channel
    pad: 2
    kernel_size: 5 #5x5 卷积核

```

group: 2 #分成 2 组, 即把 norm1 层的 96 channel 分成 2 组, conv2 层的 256 channel 分成 2 组。norm1 层中的组 1 channel 和 conv2 层中的组 1 channel 形成卷积关系, norm2 层中的组 2 channel 和 conv2 层中的组 2 channel 形成卷积关系。用这种方法可以大幅度减少卷积核的个数, 没分组前, 每个 5x5 卷积区域需要 96x256 个卷积核, 现在只需要 48x128x2 个卷积核

```

  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 1
  }
}
}

```

```

layers { #ReLU 层
  bottom: "conv2"
  top: "conv2"
  name: "relu2"
  type: RELU
}

```

```

layers { #MAX pooling 层
  bottom: "conv2"
  top: "pool2"
  name: "pool2"
  type: POOLING
  pooling_param {
    pool: MAX
    kernel_size: 3 #3x3 卷积核
    stride: 2 #步长为 2

```

```

    }
  }
  layers { #LRN层
    bottom: "pool2"
    top: "norm2"
    name: "norm2"
    type: LRN
    lrn_param {
      local_size: 5
      alpha: 0.0001
      beta: 0.75
    }
  }
  layers { #卷积层
    bottom: "norm2"
    top: "conv3"
    name: "conv3"
    type: CONVOLUTION
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
    convolution_param {
      num_output: 384 #输出 384 channel
      pad: 1
      kernel_size: 3 #3x3 卷积核
      weight_filler {
        type: "gaussian"
        std: 0.01
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
  layers { #ReLU层
    bottom: "conv3"
    top: "conv3"
    name: "relu3"
    type: RELU
  }
  layers { #卷积层
    bottom: "conv3"
    top: "conv4"
    name: "conv4"
    type: CONVOLUTION
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
  }

```

```

convolution_param {
  num_output: 384 #输出 384 channel
  pad: 1
  kernel_size: 3 #3x3 卷积核
  group: 2 #分成 2 组
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 1
  }
}
}
layers { #ReLU 层
  bottom: "conv4"
  top: "conv4"
  name: "relu4"
  type: RELU
}
layers { #卷积层
  bottom: "conv4"
  top: "conv5"
  name: "conv5"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  convolution_param {
    num_output: 256 #输出 256 channel
    pad: 1
    kernel_size: 3 #3x3 卷积核
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
}
layers { #ReLU 层
  bottom: "conv5"
  top: "conv5"
  name: "relu5"
  type: RELU
}

```



```

}
layers { #MAX pooling 层
    bottom: "conv5"
    top: "pool5"
    name: "pool5"
    type: POOLING
    pooling_param {
        pool: MAX
        kernel_size: 3
        stride: 2
    }
}
layers { #全连接层
    bottom: "pool5"
    top: "fc6"
    name: "fc6"
    type: INNER_PRODUCT
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
    inner_product_param {
        num_output: 4096 #输出 4096 个神经元
        weight_filler {
            type: "gaussian"
            std: 0.005
        }
        bias_filler {
            type: "constant"
            value: 1
        }
    }
}
layers { #ReLU 层
    bottom: "fc6"
    top: "fc6"
    name: "relu6"
    type: RELU
}
layers { #Dropout 层
    bottom: "fc6"
    top: "fc6"
    name: "drop6"
    type: DROPOUT
    dropout_param {
        dropout_ratio: 0.5 #Dropout 率为 0.5
    }
}
layers { #全连接层
    bottom: "fc6"

```

```

top: "fc7"
name: "fc7"
type: INNER_PRODUCT
blobs_lr: 1
blobs_lr: 2
weight_decay: 1
weight_decay: 0
inner_product_param {
  num_output: 4096 #输出 4096 个神经元
  weight_filler {
    type: "gaussian"
    std: 0.005
  }
  bias_filler {
    type: "constant"
    value: 1
  }
}
}
layers { #ReLU 层
  bottom: "fc7"
  top: "fc7"
  name: "relu7"
  type: RELU
}
layers { #Dropout 层
  bottom: "fc7"
  top: "fc7"
  name: "drop7"
  type: DROPOUT
  dropout_param {
    dropout_ratio: 0.5 #Dropout 率为 0.5
  }
}
layers { #全连接层
  bottom: "fc7"
  top: "fc8_pascal"
  name: "fc8_pascal"
  type: INNER_PRODUCT
  blobs_lr: 10
  blobs_lr: 20
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 21 #输出 21 个神经元, 对于 21 类, 其中物体 20 类, 背景 1 类
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"

```

```

        value: 0
    }
}
layers { #SOFTMAX 回归层
    bottom: "fc8_pascal"
    bottom: "label"
    name: "loss"
    type: SOFTMAX_LOSS
}

```

7. pascal_finetune_val.prototxt 解读

```
name: "CaffeNet"
```

```
layers {
    top: "data"
    top: "label"
    name: "data"
    type: WINDOW_DATA #WINDOW_DATA 类型的数据层
    window_data_param {

```

数据集的均值文件

```

        source: "window_file_voc_2007_test.txt"
        mean_file: "../../../data/ilsrvrc12/imagenet_mean.binaryproto" #ILSVRC12

```

```
        batch_size: 128 #batch 内有 128 张图片
```

```
        crop_size: 227 #crop 的尺寸是 227x227
```

```
        mirror: true #使用镜像图像
```

fg_threshold: 0.5 #前景的门限设置, 和 ground truth 的 overlap 大于此门限判断为识别物体 (positive 样本)

bg_threshold: 0.5 #背景的门限设置, 和 ground truth 的 overlap 小于此门限判断为背景 (negative 样本)

```
        fg_fraction: 0.25 #一个 batch 中 positive 样本数量的比例
```

```
        context_pad: 16 #clip 出的图像每边加 16 像素
```

```
        crop_mode: "warp" #crop 的方式为拉伸
```

```
    }
```

```
}
```

```
layers { #conv1 卷积层
```

```
    bottom: "data"
```

```
    top: "conv1"
```

```
    name: "conv1"
```

```
    type: CONVOLUTION
```

```
    blobs_lr: 1 #weight 的 lr 乘数
```

```
    blobs_lr: 2 #bias 的 lr 乘数
```

```
    weight_decay: 1
```

```
    weight_decay: 0
```

```
    convolution_param {
```

```
        num_output: 96 #输出 96 channel
```

```
        kernel_size: 11 #11x11 卷积核
```

```
        stride: 4 #卷积步长为 4
```

```
        weight_filler {
```



```

        type: "gaussian" #weight 高斯分布初始化
        std: 0.01
    }
    bias_filler {
        type: "constant" #bias 初始化为常数 0
        value: 0
    }
}
layers { #relu1 为 ReLU 层
    bottom: "conv1"
    top: "conv1"
    name: "relu1"
    type: RELU
}
layers { #pool1 为 MAX pooling 层
    bottom: "conv1"
    top: "pool1"
    name: "pool1"
    type: POOLING
    pooling_param {
        pool: MAX
        kernel_size: 3 #3x3 pooling
        stride: 2 #步长为 2
    }
}
layers { #norm1 为 LRN 层
    bottom: "pool1"
    top: "norm1"
    name: "norm1"
    type: LRN
    lrn_param {
        local_size: 5 #5 个连续特征点做 LRN
        alpha: 0.0001 #LRN 公式中的 alpha 参数
        beta: 0.75 #LRN 公式中的 beta 参数
    }
}
layers { #conv2 为卷积层
    bottom: "norm1"
    top: "conv2"
    name: "conv2"
    type: CONVOLUTION
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
    convolution_param {
        num_output: 256 #输出为 256 channel
        pad: 2
        kernel_size: 5 #5x5 卷积核
    }
}

```

group: 2 #分成2组, 即把 norm1 层的 96 channel 分成 2 组, conv2 层的 256 channel 分成 2 组。norm1 层中的组 1 channel 和 conv2 层中的组 1 channel 形成卷积关系, norm2 层中的组 2 channel 和 conv2 层中的组 2 channel 形成卷积关系。用这种方法可以大幅度减少卷积核的个数, 没分组前, 每个 5x5 卷积区域需要 96x256 个卷积核, 现在只需要 48x128x2 个卷积核

```

    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
layers { #ReLU层
  bottom: "conv2"
  top: "conv2"
  name: "relu2"
  type: RELU
}
layers { #MAX pooling层
  bottom: "conv2"
  top: "pool2"
  name: "pool2"
  type: POOLING
  pooling_param {
    pool: MAX
    kernel_size: 3 #3x3 卷积核
    stride: 2 #步长为 2
  }
}
layers { #LRN层
  bottom: "pool2"
  top: "norm2"
  name: "norm2"
  type: LRN
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
layers {#卷积层
  bottom: "norm2"
  top: "conv3"
  name: "conv3"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
}
```

```

weight_decay: 0
convolution_param {
  num_output: 384 #输出 384 channel
  pad: 1
  kernel_size: 3 #3x3 卷积核
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
}
layers { #ReLU 层
  bottom: "conv3"
  top: "conv3"
  name: "relu3"
  type: RELU
}
layers { #卷积层
  bottom: "conv3"
  top: "conv4"
  name: "conv4"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  convolution_param {
    num_output: 384 #输出 384 channel
    pad: 1
    kernel_size: 3 #3x3 卷积核
    group: 2 #分成 2 组
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
}
layers { #ReLU 层
  bottom: "conv4"
  top: "conv4"
  name: "relu4"
  type: RELU
}

```



```

}
layers { #卷积层
  bottom: "conv4"
  top: "conv5"
  name: "conv5"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  convolution_param {
    num_output: 256 #输出 256 channel
    pad: 1
    kernel_size: 3 #3x3 卷积核
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
layers { #ReLU层
  bottom: "conv5"
  top: "conv5"
  name: "relu5"
  type: RELU
}
layers { #MAX pooling层
  bottom: "conv5"
  top: "pool5"
  name: "pool5"
  type: POOLING
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layers { #全连接层
  bottom: "pool5"
  top: "fc6"
  name: "fc6"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0

```

```

    inner_product_param {
      num_output: 4096 #输出 4096 个神经元
      weight_filler {
        type: "gaussian"
        std: 0.005
      }
      bias_filler {
        type: "constant"
        value: 1
      }
    }
  }
}
layers { #ReLU 层
  bottom: "fc6"
  top: "fc6"
  name: "relu6"
  type: RELU
}
layers { #Dropout 层
  bottom: "fc6"
  top: "fc6"
  name: "drop6"
  type: DROPOUT
  dropout_param {
    dropout_ratio: 0.5 #Dropout 率为 0.5
  }
}
layers { #全连接层
  bottom: "fc6"
  top: "fc7"
  name: "fc7"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 4096 #输出 4096 个神经元
    weight_filler {
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
}
layers { #ReLU 层
  bottom: "fc7"

```

```

    top: "fc7"
    name: "relu7"
    type: RELU
  }
  layers { #Dropout层
    bottom: "fc7"
    top: "fc7"
    name: "drop7"
    type: DROPOUT
    dropout_param {
      dropout_ratio: 0.5 #Dropout 率为0.5
    }
  }
  layers { #全连接层
    bottom: "fc7"
    top: "fc8_pascal"
    name: "fc8_pascal"
    type: INNER_PRODUCT
    blobs_lr: 10
    blobs_lr: 20
    weight_decay: 1
    weight_decay: 0
    inner_product_param {
      num_output: 21 #输出 21 个神经元, 对于 21 类, 其中物体 20 类, 背景 1 类
      weight_filler {
        type: "gaussian"
        std: 0.01
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
}
layers { #SOFTMAX 层, 用于输出 21 类概率
  bottom: "fc8_pascal"
  top: "prob"
  name: "prob"
  type: SOFTMAX
}
layers { #accuracy 层, 输出预测和 ground truth 相比后的准确率
  bottom: "prob"
  bottom: "label"
  top: "accuracy"
  name: "accuracy"
  type: ACCURACY
}

```


参考文献

- [1] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In CVPR, 2014.
- [2] R. Girshick. Fast R-CNN. arXiv:1504.08083, 2015.
- [3] Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards real-time object detection with region proposal networks. In: NIPS. (2015).
- [4] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders. Selective search for object recognition. IJCV, 2013.

13

第 13 章 Fast-RCNN 模型

在本书前面的章节中,我们介绍了深度学习在目标探测领域的经典之作 R-CNN^[1]模型。继 2014 年的 R-CNN 之后, R-CNN 的原作者 Ross Girshick 再一次推出了 R-CNN 的继承者 Fast-RCNN^[2], Fast-RCNN 在继承 R-CNN 的同时,吸取了 SPPnet^[3]的特点,构思精巧,流程更为紧凑,大幅提升了目标检测的速度。

13.1 Fast-RCNN 模型简介

Fast-RCNN 和 R-CNN 相比,训练时间从 84 小时减少为 9.5 小时,测试时间从 47 秒减少为 0.32 秒。在 PASCAL VOC 2007 上的准确率约在 66%~67%之间。另外,在训练速度上,比 SPPnet 快 3 倍,在测试速度上,比 SPPnet 快 10 倍。

Fast-RCNN 方法解决了 R-CNN 方法三个问题。

- 问题一: 测试时速度慢

R-CNN 把一张图像分解成大量的建议框,每个建议框拉伸形成的图像都会单独通过 CNN 提取特征。实际上这些建议框之间大量重叠,特征值之间完全可以共享,造成了运算能力的浪费。

Fast-RCNN 改进了这个问题。Fast-RCNN 将整张图像归一化后直接送入 CNN。在最后的卷积层输出的 feature map 上, 加入建议框信息, 使得在此之前的 CNN 运算得以共享。

- 问题二: 训练时速度慢

R-CNN 在训练时, 是在采用 SVM 分类之前, 把通过 CNN 提取的特征存储在硬盘上。这种方法造成了训练性能低下, 因为在硬盘上大量的读写数据会造成训练速度缓慢。

Fast-RCNN 在训练时, 只需要将一张图像送入网络, 每张图像一次性地提取 CNN 特征和建议区域, 训练数据在 GPU 内存里直接进 Loss 层, 这样候选区域的前几层特征不需要再重复计算且不再需要把大量数据存储在硬盘上。

- 问题三: 训练所需空间大

R-CNN 中独立的 SVM 分类器和回归器需要大量特征作为训练样本, 需要大量的硬盘空间。Fast-RCNN 把类别判断和位置回归统一用深度网络实现, 不再需要额外存储。

Fast-RCNN 和 R-CNN 一样, 是深度学习算法在目标检测任务上的重大进步, 让业界看到了深度学习算法在目标检测任务上进行实时处理的希望。

13.2 Fast-RCNN 的特点和使用场景

Fast-RCNN 是深度学习在目标检测任务上的应用, Fast 是相对 R-CNN 模型而言, 是 R-CNN 的加速版本。

整体训练框架大致为:

- 生成建议窗口 (Selective Search^[4]), Fast-RCNN 用 Selective Search 方法生成建议窗口 (proposals), 每张图片大约生成 2000 个建议窗口;
- Fast-RCNN 把整张图片输入 CNN, 进行特征提取;
- Fast-RCNN 把建议窗口映射到 CNN 的最后一层卷积 feature map 上;

- 通过 RoI pooling 层使每个建议窗口生成固定尺寸的 feature map;
- 利用 Softmax Loss 和 Smooth L1 Loss 对分类概率和边框回归 (Bounding box regression) 联合训练。

整体探测框架大致为:

- 生成建议窗口, Fast-RCNN 用 Selective Search 方法生成建议窗口 (proposals), 每张图片大约生成 2000 个建议窗口;
- Fast-RCNN 把整张图片输入 CNN, 进行特征提取;
- Fast-RCNN 把建议窗口映射到 CNN 的最后一层卷积 feature map 上;
- 通过 RoI pooling 层使每个建议窗口生成固定尺寸的 feature map;
- 利用 Softmax Loss 探测分类概率;
- 利用 Smooth L1 Loss 探测边框回归;
- 用边框回归值校正原来的建议窗口, 生成预测窗口坐标。

由于用于目标检测的数据集较小, 无法充分从零开始训练网络, Fast-RCNN 采用在预训练网络上微调的方法。即通过 ILSVRC 2012 数据集对 CNN 进行预训练, 然后用 PASCL VOC 数据集再进行微调训练。Fast-RCNN 提供了三种预训练网络模型, 分别是小型网络 CaffeNet、中型网络 VGG_CNN_M_1024、大型网络 VGG16。图 13-1 是 Fast-RCNN 的模型结构图。

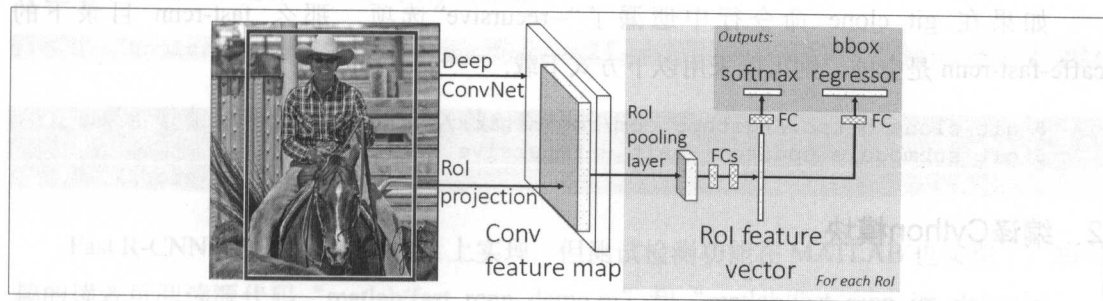


图 13-1 Fast-RCNN 模型结构图

表 13-1 是 Fast-RCNN 在各数据集上的表现。

表 13-1 Fast-RCNN 在各数据集上的表现

模型	VOC2007 mAP	VOC2012 mAP
SPPnet bbox reg	63.1	
R-CNN bbox reg	66.0	62.9
Fast-RCNN	66.9	66.1

Fast-RCNN 相比于 R-CNN 而言, 不仅在速度有大幅度提高, 而且 mAP 也有一定程度的上升。

13.3 Caffe Fast-RCNN 解读

Fast-RCNN 基于 Caffe 框架的全部实现在 Github 上是开源的。我们可以按以下步骤来 DIY 训练 Fast-RCNN。

13.3.1 Fast-RCNN 模型训练准备

1. 下载Fast-RCNN源代码并安装

可以通过 git 工具直接下载, 在 Linux 命令行下输入:

```
$ git clone --recursive https://github.com/rbgirshick/fast-rcnn.git
```

如果在 git clone 命令行中遗漏了“--recursive”选项, 那么 fast-rcnn 目录下的 caffe-fast-rcnn 是空的, 则可以采用以下方式下载:

```
$ git clone https://github.com/rbgirshick/fast-rcnn.git
$ git submodule update --init --recursive
```

2. 编译Cython模块

在 fast-rcnn/lib 目录下, 运行以下命令:


```
$ make
```

注意: Cython 可以把 Python 代码编译成 C 代码, 再用 GCC 编译成二进制代码, 可大幅度提高 Python 代码的执行速度。

3. 编译Caffe和Pycaffe

```
$ make
$ make pycaffe
```

注意: 由于 Fast-RCNN 使用 Python 层, 请读者在 Makefile.config 中把 WITH_PYTHON_LAYER 设为 1。

4. 下载训练过的Fast R-CNN detectors

在 fast_rcnn 目录下执行以下命令:

```
$ ./data/scripts/fetch_fast_rcnn_models.sh
```

5. 运行基于python的demo

在 fast_rcnn 目录下执行以下命令:

```
$ ./tools/demo.py
```

Demo 采用事先训练好的 VGG16 模型, 目标建议框也是事先计算好的。

注意: 如果 demo 运行 Caffe 时崩溃, 很可能是由于 GPU 没有足够大的内存。这时请采用较小的模型来运行 Demo, 执行命令如下:

```
$ ./tools/demo.py --net vgg_cnn_m_1024
```

如果希望在 CPU 上运行 Demo, 执行命令如下:

```
$ ./tools/demo.py --cpu
```

Fast R-CNN 的训练只在 Python 上实现, 但测试检测功能在 MATLAB 也实现了。感兴趣的读者可阅读源代码 “matlab/fast_rcnn_demo.m” 和 “matlab/fast_rcnn_im_detect.m”,

了解实现细节。

6. 计算建议框

Demo 采用 selective search proposals 计算建议框, 感兴趣的读者可通过以下链接下载: https://github.com/rbgirshick/rcnn/blob/master/selective_search/selective_search_boxes.m。

另外, 计算建议框有很多其他选择, 下面列举了一些计算建议框的方法:

- EdgeBoxes: [matlab code](<https://github.com/pdollar/edges>)
- GOP and LPO: [python code](<http://www.philkr.net/>)
- MCG: [matlab code](<http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/mcg/>)
- RIGOR: [matlab code](<http://cpl.cc.gatech.edu/projects/RIGOR/>)

13.3.2 Fast-RCNN 模型训练

1. 训练过程

如果读者对训练感兴趣, 可按下列步骤 DIY 训练。

(1) 下载训练、测试数据集, 执行以下命令:

```
$wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtrainval_06-Nov-2007.tar
$wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtest_06-Nov-2007.tar
$wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCdevkit_08-Jun-2007.tar
$tar xvf VOCtrainval_06-Nov-2007.tar
$tar xvf VOCtest_06-Nov-2007.tar
$tar xvf VOCdevkit_08-Jun-2007.tar
```

上述命令执行完毕后, 应有以下结构:

```
$VOCdevkit/           # development kit
$VOCdevkit/VOCcode/    # VOC utility code
$VOCdevkit/VOC2007     # image sets, annotations, etc.
# ... and several other directories ...
```

(3) 为 PASCAL VOC 数据集创建 symlinks:

```
$cd data
$ln -s $VOCdevkit VOCdevkit2007
```

因为 PASCAL 数据集可能被多个项目使用, 用 symlinks 会非常方便。

(4) 用相同的步骤获取 PASCAL VOC 2010 和 2012。

(5) 下载预先计算好的 VOC2007 和 VOC2012 数据集的 selective search boxes 可通过执行下面命令获取。

```
$ ./data/scripts/fetch_selective_search_data.sh
```

(6) 下载已训练 ImageNet models。

在论文中提到的已训练 ImageNet (包括 CaffeNet, VGG_CNN_M_1024, VGG16) 模型可通过执行下面命令获取:

```
$ ./data/scripts/fetch_imagenet_models.sh
```

读者也可以到 github 的 Caffe Model-Zoo 里获取这些模型。链接如下: <https://github.com/BVLC/caffe/wiki/Model-Zoo>。

(7) 运行以下命令开始训练。

```
$ ./tools/train_net.py --gpu 0 --solver models/VGG16/solver.prototxt \
  --weights data/imagenet_models/VGG16.v2.caffemodel
```

如果有以下错误输出:

```
EnvironmentError: MATLAB command 'matlab' not found. Please add 'matlab' to
your PATH.
```

这说明 matlab 的路径没有加到 \$PATH 中, 而训练过程中用到了 MATLAB。

运行以下命令测试已训练的模型，测试结果输出到./output 目录中。

```
$ ./tools/test_net.py --gpu 1 --def models/VGG16/test.prototxt \
  --net output/default/voc_2007_trainval/vgg16_fast_rcnn_iter_40000.
caffemodel
```

可通过以下命令使用论文中提到的 truncated SVD 压缩 Fast-RCNN 模型：

```
$ ./tools/compress_net.py --def models/VGG16/test.prototxt \
  --def-svd models/VGG16/compressed/test.prototxt \
  --net output/default/voc_2007_trainval/vgg16_fast_rcnn_iter_40000.
caffemodel
```

可使用以下命令测试已压缩的模型：

```
$ ./tools/test_net.py --gpu 0 --def models/VGG16/compressed/test.prototxt \
  --net output/default/voc_2007_trainval/vgg16_fast_rcnn_iter_40000_
svd_fc6_1024_fc7_256.caffemodel
```

2. Fast-rcnn caffe模型的solver.prototxt解读

```
train_net: "models/CaffeNet/train.prototxt" #训练用的网络 prototxt 文件
base_lr: 0.001 #基本学习率
lr_policy: "step" #step 类型的 lr 策略，具体的 step 由 stepsize 决定
gamma: 0.1
stepsize: 30000 #每 30000 次迭代时改变 base_lr
display: 20 #每 20 次迭代显示一次迭代信息
average_loss: 100
momentum: 0.9
weight_decay: 0.0005
# We disable standard caffe solver snapshotting and implement our own snapshot
# function
#作者用自己开发的 snapshot 功能，所以这里设为 0
snapshot: 0
# We still use the snapshot prefix, though
snapshot_prefix: "caffenet_fast_rcnn"
#debug_info: true
```

3. Fast-rcnn caffe模型的train.prototxt解读

```
name: "CaffeNet"
layer {
  name: 'data'
  type: 'Python' #原作者自己实现的 Python 层
  top: 'data' #图像数据
```

```

top: 'rois' #RoI 数据 (Selective Search 计算所得建议框)
top: 'labels' #标签
top: 'bbox_targets' #Bounding Box 目标
top: 'bbox_loss_weights' #Bounding Box 回归的 loss 的权重
python_param {
    module: 'roi_data_layer.layer' #Python 层的模块
    layer: 'RoIDataLayer' #Python 层的实现
    param_str: "'num_classes': 21" #一共识别 21 类, 包括背景
}
}
layer {
    name: "conv1" #卷积层
    type: "Convolution"
    bottom: "data"
    top: "conv1"
    param {
        lr_mult: 0
        decay_mult: 0
    }
    param {
        lr_mult: 0
        decay_mult: 0
    }
    convolution_param {
        num_output: 96 #输出 96 个 channel
        kernel_size: 11 #卷积核的尺寸为 11x11
        pad: 5 #每边加 5 个像素的 pad
        stride: 4 #步长为 4
    }
}
layer {
    name: "relu1" #ReLU 层
    type: "ReLU"
    bottom: "conv1"
    top: "conv1"
}
layer {
    name: "pool1" #MAX Pooling 层
    type: "Pooling"
    bottom: "conv1"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 3
        pad: 1
        stride: 2
    }
}
layer {

```



```

name: "norm1" #LRN 层
type: "LRN"
bottom: "pool1"
top: "norm1"
lrn_param {
  local_size: 5
  alpha: 0.0001
  beta: 0.75
}
}
layer {
  name: "conv2" #卷积层
  type: "Convolution"
  bottom: "norm1"
  top: "conv2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256 #输出为 256 个 channel
    kernel_size: 5 #5x5 卷积核
    pad: 2
    group: 2
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "conv2"
  top: "conv2"
}
layer {
  name: "pool2" #MAX Pooling 层
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 3
    pad: 1
    stride: 2
  }
}
layer {
  name: "norm2" #LRN 层
  type: "LRN"

```

```

bottom: "pool2"
top: "norm2"
lrn_param {
  local_size: 5
  alpha: 0.0001
  beta: 0.75
}
}
layer {
  name: "conv3" #卷积层
  type: "Convolution"
  bottom: "norm2"
  top: "conv3"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384 #输出为 384 channel
    kernel_size: 3 #3x3 卷积核
    pad: 1
  }
}
}
layer {
  name: "relu3" #ReLU 层
  type: "ReLU"
  bottom: "conv3"
  top: "conv3"
}
}
layer {
  name: "conv4" #卷积层
  type: "Convolution"
  bottom: "conv3"
  top: "conv4"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384 #输出为 384 channel
    kernel_size: 3 #3x3 卷积核
    pad: 1
  }
}

```

```

        group: 2 #分组, 降低参数数量
    }
}
layer {
    name: "relu4" #ReLU 层
    type: "ReLU"
    bottom: "conv4"
    top: "conv4"
}
layer {
    name: "conv5" #卷积层
    type: "Convolution"
    bottom: "conv4"
    top: "conv5"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256 #输出为 256 channel
        kernel_size: 3 #3x3 卷积核
        pad: 1
        group: 2 #分组
    }
}
layer {
    name: "relu5" #ReLU 层
    type: "ReLU"
    bottom: "conv5"
    top: "conv5"
}
layer {
    name: "roi_pool5" #ROI Pooling, 这个层为原作者在 Fast-RCNN 中对 Caffe 的扩展,
    type: "ROIPooling"
    bottom: "conv5"
    bottom: "rois"
    top: "pool5"
    roi_pooling_param {
        pooled_w: 6
        pooled_h: 6
        spatial_scale: 0.0625 # 1/16 #经过前面的 conv1 层的 (尺寸为原来 1/4) 和两次
        MAX Pooling 层 (尺寸为原来的 1/2), 所以到 ROI Pooling 层时, 所有的尺寸都为原图的 1/16
    }
}
layer {

```



```

name: "fc6" #全连接层
type: "InnerProduct"
bottom: "pool5"
top: "fc6"
param {
  lr_mult: 1
  decay_mult: 1
}
param {
  lr_mult: 2
  decay_mult: 0
}
inner_product_param {
  num_output: 4096 #输出 4096 个神经元
}
}
layer {
  name: "relu6" #ReLU 层
  type: "ReLU"
  bottom: "fc6"
  top: "fc6"
}
layer {
  name: "drop6" #Dropout 层
  type: "Dropout"
  bottom: "fc6"
  top: "fc6"
  dropout_param {
    dropout_ratio: 0.5 #Dropout 率 0.5
  }
}
layer {
  name: "fc7" #全连接层
  type: "InnerProduct"
  bottom: "fc6"
  top: "fc7"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 4096 #输出 4096 个神经元
  }
}
layer {
  name: "relu7" #ReLU 层

```

```

    type: "ReLU"
    bottom: "fc7"
    top: "fc7"
}
layer {
    name: "drop7" #Dropout 层
    type: "Dropout"
    bottom: "fc7"
    top: "fc7"
    dropout_param {
        dropout_ratio: 0.5 #Dropout 率0.5
    }
}
layer {
    name: "cls_score" #全连接层
    type: "InnerProduct"
    bottom: "fc7"
    top: "cls_score"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    inner_product_param {
        num_output: 21 #输出 21 个神经元, 对应 21 类
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}
layer {
    name: "bbox_pred" #全连接层
    type: "InnerProduct"
    bottom: "fc7"
    top: "bbox_pred"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
}

```


inner_product_param {
 num_output: 84 #输出 84 个神经元, 对应 21 类, 每类的 Bounding Box 的回归值为 $\langle x, y, w, h \rangle$, 共 4 个数值, 所以一共 $21 \times 4 = 84$ 。注意, 这里原作者把不同类别的 Bounding Box 做分别预测, 是因为考虑到不同类别的物体差别较大, 所有回归值也应差别较大。不过在后面介绍的 SSD 模型中, SSD 的原作者对所有物体类别采用共享的回归值, 也获得了很高的 mAP。

```
weight_filler {
  type: "gaussian"
  std: 0.001
}
```

```
bias_filler {
  type: "constant"
  value: 0
}
```

```
}
}
```

#下面 Fast-RCNN 把两个回归进行联合, 来训练网络

总 Loss = SoftMax Loss + bbox_loss_weights * SmoothL1Loss

```
layer {
  name: "loss_cls" #SoftMax Loss 回归
  type: "SoftmaxWithLoss"
  bottom: "cls_score"
  bottom: "labels"
  top: "loss_cls"
  loss_weight: 1
}
```

```
layer {
  name: "loss_bbox" #SmoothL1Loss 回归
  type: "SmoothL1Loss"
  bottom: "bbox_pred"
  bottom: "bbox_targets"
  bottom: "bbox_loss_weights"
  top: "loss_bbox"
  loss_weight: 1
}
```

4. 测试用test.prototxt解读

test.prototxt 的输入和输出和 train.prototxt 不同, 其他的都一样, 这里着重注释一下 test.prototxt 的输入和输出。

```
name: "CaffeNet"
input: "data"
input_shape {
  dim: 1 #测试为输入 1 张图像
  dim: 3 #RGB 3 通道
  dim: 227 #227x227
  dim: 227
}
input: "rois" #Selective Search 产生的 RoI
```

```

input_shape {
  dim: 1 # to be changed on-the-fly to num ROIs
  dim: 5 # [batch ind, x1, y1, x2, y2] zero-based indexing
}
layer {
  name: "conv1" #卷积层
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  convolution_param {
    num_output: 96 #输出 96 个 channel
    kernel_size: 11 #卷积核的尺寸为 11x11
    pad: 5 #每边加 5 个像素的 pad
    stride: 4 #步长为 4
  }
}
layer {
  name: "relu1" #ReLU 层
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "pool1" #MAX Pooling 层
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    pad: 1
    stride: 2
  }
}
layer {
  name: "norm1" #LRN 层
  type: "LRN"
  bottom: "pool1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
  }
}

```

```

    beta: 0.75
  }
}
layer {
  name: "conv2" #卷积层
  type: "Convolution"
  bottom: "norm1"
  top: "conv2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256 #输出为 256 个 channel
    kernel_size: 5 #5x5 卷积核
    pad: 2
    group: 2
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "conv2"
  top: "conv2"
}
layer {
  name: "pool2" #MAX Pooling 层
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 3
    pad: 1
    stride: 2
  }
}
layer {
  name: "norm2" #LRN 层
  type: "LRN"
  bottom: "pool2"
  top: "norm2"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}

```



```

}
layer {
  name: "conv3" #卷积层
  type: "Convolution"
  bottom: "norm2"
  top: "conv3"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384 #输出为 384 channel
    kernel_size: 3 #3x3 卷积核
    pad: 1
  }
}
layer {
  name: "relu3" #ReLU 层
  type: "ReLU"
  bottom: "conv3"
  top: "conv3"
}
layer {
  name: "conv4" #卷积层
  type: "Convolution"
  bottom: "conv3"
  top: "conv4"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384 #输出为 384 channel
    kernel_size: 3 #3x3 卷积核
    pad: 1
    group: 2 #分组, 降低参数数量
  }
}
layer {
  name: "relu4" #ReLU 层
  type: "ReLU"
  bottom: "conv4"

```

```

    top: "conv4"
  }
  layer {
    name: "conv5" #卷积层
    type: "Convolution"
    bottom: "conv4"
    top: "conv5"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    convolution_param {
      num_output: 256 #输出为 256 channel
      kernel_size: 3 #3x3 卷积核
      pad: 1
      group: 2 #分组
    }
  }
  layer {
    name: "relu5" #ReLU 层
    type: "ReLU"
    bottom: "conv5"
    top: "conv5"
  }
  layer {
    name: "roi_pool5" #ROI Pooling, 这个层为原作者在 Fast-RCNN 中对 Caffe 的扩展,
    #使得所有的 RoI 通过此层获的相同的尺寸 (6x6), 以便后面全连接层的处理
    type: "ROIPooling"
    bottom: "conv5"
    bottom: "rois"
    top: "pool5"
    roi_pooling_param {
      pooled_w: 6
      pooled_h: 6
      spatial_scale: 0.0625 # 1/16 #经过前面的 conv1 层的 (尺寸为原来 1/4) 和两次
      #MAX Pooling 层 (尺寸为原来的 1/2), 所以到 ROI Pooling 层时, 所有的尺寸都为原图的 1/16
    }
  }
  layer {
    name: "fc6" #全连接层
    type: "InnerProduct"
    bottom: "pool5"
    top: "fc6"
    param {
      lr_mult: 1
      decay_mult: 1
    }
  }

```



```

    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    inner_product_param {
      num_output: 4096 #输出 4096 个神经元
    }
  }
  layer {
    name: "relu6" #ReLU 层
    type: "ReLU"
    bottom: "fc6"
    top: "fc6"
  }
  layer {
    name: "drop6" #Dropout 层
    type: "Dropout"
    bottom: "fc6"
    top: "fc6"
    dropout_param {
      dropout_ratio: 0.5 #Dropout 率 0.5
    }
  }
  layer {
    name: "fc7" #全连接层
    type: "InnerProduct"
    bottom: "fc6"
    top: "fc7"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    inner_product_param {
      num_output: 4096 #输出 4096 个神经元
    }
  }
  layer {
    name: "relu7" #ReLU 层
    type: "ReLU"
    bottom: "fc7"
    top: "fc7"
  }
  layer {
    name: "drop7" #Dropout 层
    type: "Dropout"
  }

```

```

bottom: "fc7"
top: "fc7"
dropout_param {
  dropout_ratio: 0.5 #Dropout 率0.5
}
}
layer {
  name: "cls_score" #全连接层
  type: "InnerProduct"
  bottom: "fc7"
  top: "cls_score"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 21 #输出 21 个神经元, 对应 21 类
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
}
layer {
  name: "bbox_pred" #全连接层
  type: "InnerProduct"
  bottom: "fc7"
  top: "bbox_pred"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 84 #输出 84 个神经元, 对应 21 类, 每类的 Bounding Box 的回归值为
    <x,y,w,h>, 共 4 个数值, 所以一共 21x4=84。注意, 这里原作者把不同类别的 Bounding Box 做分
    别预测, 是因为考虑到不同类别的物体差别较大, 所有回归值也应差别较大。不过在后面介绍的 SSD 模
    型中, SSD 的原作者对所有物体类别采用共享的回归值, 也获得了很高的 mAP。
    weight_filler {

```

```
        type: "gaussian"
        std: 0.001
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
  layer {
    name: "cls_prob" #Softmax 层, 输入每一类的概率
    type: "Softmax"
    bottom: "cls_score"
    top: "cls_prob"
  }
}
```

参考文献

- [1] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In CVPR, 2014.
- [2] R. Girshick. Fast R-CNN. arXiv:1504.08083, 2015.
- [3] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In ECCV, 2014.
- [4] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders. Selective search for object recognition. IJCV, 2013.

14

第 14 章 Faster-RCNN 模型

在本书前面的章节中，我们分别介绍了深度学习在目标探测领域的经典之作 R-CNN^[1]模型和 Fast-RCNN^[2]模型。虽然 Fast-RCNN 基于 R-CNN 有很大的改进，但 Fast-RCNN 仍然基于 Selective Search^[3]方法提取建议框，而 Selective Search 方法提取建议框的计算是基于 CPU 运行的，无法借用 GPU 的高度并行运算能力，所以效率很低。另外这种方法为每幅图像提供大约 2000 个建议框。由于建议框过多，也加重了后面深度学习的处理压力。Faster-RCNN^[4]在吸取了 Fast-RCNN 的特点的前提下，采用共享的卷积网组成 RPN (Region Proposal Network)，用 RPN 直接预测出建议框，数据限定在 300 个，RPN 的预测绝大部分在 GPU 中完成，且卷积网和 Fast-RCNN 部分共享，因此大幅提升了目标检测的速度。本章介绍了 Faster-RCNN^[4]模型的特点，并做具体的解读。

14.1 Faster-RCNN 模型简介

Faster-RCNN 采用共享的卷积网组成 RPN (Region Proposal Network)，用 RPN 直接预测出建议框，数据限定在 300 个。由于 RPN 预测出的建议框质量高、数量少，且 RPN 的预测绝大部分在 GPU 中完成，同时卷积网和 Fast-RCNN 部分共享，因此大幅提升了目标检测的速度。

Faster-RCNN 把目标检测的四个基本步骤（候选区域生成、特征提取、分类和 Bounding Box 回归）统一到一个深度网络框架之内。其中候选区域生成由 RPN 部分完成，是 Faster-RCNN 的创意所在。特征提取、分类和 Bounding Box 回归等三个部分还是沿用了 Fast-RCNN。所以说 Faster-RCNN 是 Fast-RCNN 的升级版。

Faster-RCNN 在各数据集上的表现如表 14-1 和表 14-2 所示。

表 14-1 Faster-RCNN 的不同模型基于不同建议框提取方法的表现

模型	建议框提取方法	每幅图像处理速度	每秒帧数
VGG-16	Selective Search	1830ms	0.5 fps
VGG-16	RPN	198ms	5 fps
ZF	RPN	59ms	17 fps

表 14-2 Faster-RCNN 在各数据集上的表现

模型	训练数据集	测试数据集	mAP	每幅图像处理速度
Faster RCNN, VGG-16	VOC 2007 trainval	VOC 2007 test	69.9	198ms
Faster RCNN, VGG-16	VOC 2007 trainval + 2012 trainval	VOC 2007 test	73.2	198ms
Faster RCNN, VGG-16	VOC 2012 trainval	VOC 2012 test	67.0	198ms
Faster RCNN, VGG-16	VOC 2007 trainval&test + 2012 trainval	VOC 2012 test	70.4	198ms

Faster-RCNN 可以简单地看做是 PRN（区域生成网络）和 Fast-RCNN 组合而成，用 RPN 代替 Fast-RCNN 中的 Selective Search 方法是 Faster-RCNN 的核心思想。

Faster-RCNN 和 Fast-RCNN 相比，在 PASCAL VOC 2007 上的准确率略有提高。图像的处理速度大幅度提高。在测试上，Faster-RCNN 比 Fast-RCNN 快 10 倍，如果采用 ZF 模型，图像处理速度达到 17fps，就能达到准实时处理的能力，这是一个巨大的进步。

Faster-RCNN 方法解决了 Fast-RCNN 方法的一个疑难问题，如何高效快速地产生建议框？Faster-RCNN 创造性地采用卷积网络自行产生建议框，并且和目标检测网络共享卷积网络，使得建议框数目从原有的约 2000 个减少为 300 个，且建议框的质量也有本质的提高。

Faster-RCNN 是深度学习算法在目标检测任务上的重大进步，让业界看到了深度学习算法在目标检测任务上进行实时处理的希望。有兴趣的读者可参阅论文“Faster R-CNN:

Towards Real-Time Object Detection with Region Proposal Networks”。

14.2 Faster-RCNN 的特点和使用场景

Faster-RCNN 是深度学习在目标检测任务上的应用，Faster 是相对 Fast-RCNN 模型而言的，是 RCNN 系列的最优版本。

Faster-RCNN 有两个关键点：一是使用 RPN 代替原来的 Selective Search 方法产生建议窗口；二是产生建议窗口的 CNN 和目标检测的 CNN 共享。

整体框架大致为：

- Faster-RCNN 把整张图片输入 CNN，进行特征提取；
- 生成建议窗口（RPN），Faster-RCNN 用 RPN 生成建议窗口（proposals），每张图片生成 300 个建议窗口；
- Faster-RCNN 把建议窗口映射到 CNN 的最后一层卷积 feature map 上；
- 通过 RoI pooling 层使每个 RoI 生成固定尺寸的 feature map；
- 利用 Softmax Loss 和 Smooth L1 Loss 对分类概率和边框回归（Bounding box regression）联合训练。

测试时，用边框回归值校正原来的候选窗口，生成预测窗口坐标。

Faster-RCNN 的训练和 Fast-RCNN 的训练有所区别，主要是为了生成建议框的 RPN 和探测物体的网络共用 CNN。在论文中，原作者提出了分阶段训练的方法。在代码上，作者又提供了一种速度更快的端对端训练的方法。

Faster-RCNN 提供了三种预训练网络模型作为基本网络，分别是小型网络 ZF、中型网络 VGG_CNN_M_1024 和大型网络 VGG16^[5]。

在生成建议窗口的 RPN 设计中, Faster-RCNN 在 feature map 上运用滑动窗口。为了能够应对不同尺寸的物体, Faster-RCNN 采用了 3 种不同类型的滑动窗口 (Anchor), 长宽比是 1: 1、2: 1、1: 2, 并用 3 个尺度缩放滑动窗口, 一共采用了 9 种类型的滑动窗口。这些窗口经过卷积形成 256 维向量, 最终通过分类挑选出 300 个得分最高的窗口作为最终的建议窗口。图 14-1 是 PRN 模型的结构图。

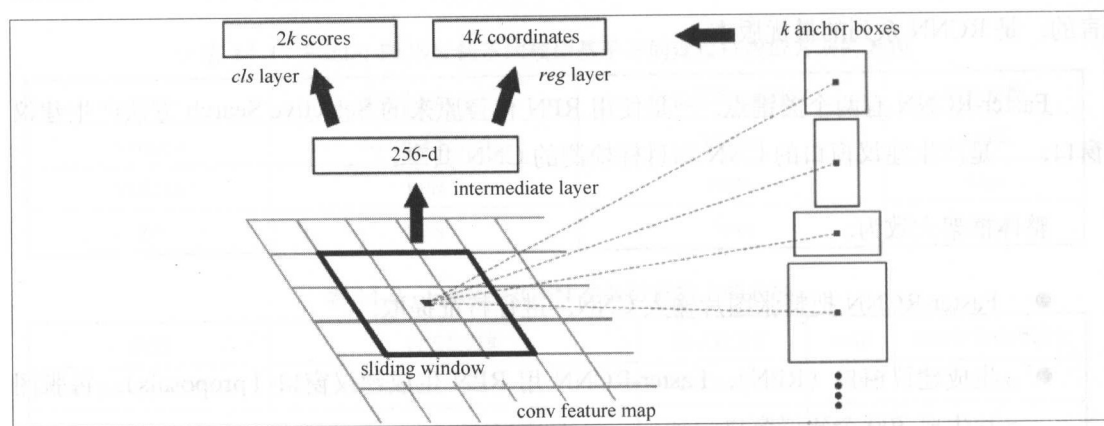


图 14-1 RPN 模型结构图

Faster-RCNN 相比于 Fast-RCNN 而言, 建议窗口的产生更快并且质量很高。

14.3 Caffe Faster-RCNN 解读

Faster-RCNN 基于 Caffe 框架的全部实现在 Github 上是开源的。在 Github 上, 可以找到 Faster-RCNN 的两个版本: 一个是基于 MATLAB 的, 另一个是基于 Python 的。本书选择讲解的是基于 Python 的版本。可以按以下步骤来 DIY 训练 Faster-RCNN。

14.3.1 Faster-RCNN 模型训练准备

1. 下载Faster-RCNN源代码并安装

通过 git 工具直接下载, 在 Linux 命令行输入:

```
$ git clone --recursive https://github.com/rbgirshick/pyfaster-rcnn.git
```

如果在 git clone 时遗漏了“--recursive”，那么 faster-rcnn 目录下的 caffe-faster-rcnn 可能是空的，可以采用以下方式下载：

```
$ git clone https://github.com/rbgirshick/py-faster-rcnn.git
$ git submodule update --init --recursive
```

2. 编译Cython模块

在 Faster-RCNN/lib 目录下，运行以下命令：

```
$ make
```

注意：Cython 可以把 Python 代码编译成 C 代码，再用 GCC 编译，可大幅度提高 Python 代码的执行速度。

3. 编译Caffe和Pycaffe

```
$make
$make pycaffe
```

注意：由于 Fast-RCNN 使用 Python 层，请读者在 Makefile.config 中把 WITH_PYTHON_LAYER 设为 1。

4. 下载事先训练过的Faster R-CNN detectors

执行以下命令：

```
$ ./data/scripts/fetch_faster_rcnn_models.sh
```

5. 运行基于python的demo

执行以下命令：

```
$./tools/demo.py
```

Demo 采用事先训练好的 VGG16 模型，和 Fast-RCNN 的 demo 不同的是，这个 Demo

的目标建议框是执行 Demo 时实时计算出来的。

14.3.2 Faster-RCNN 模型训练

1. 训练过程

如果读者对训练感兴趣，可按下列步骤 DIY 训练。

(1) 下载训练、测试数据集。

执行以下命令：

```
$wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtrainval_06-Nov-2007.tar
$wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtest_06-Nov-2007.tar
$wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCdevkit_08-Jun-2007.tar
$tar xvf VOCtrainval_06-Nov-2007.tar
$tar xvf VOCtest_06-Nov-2007.tar
$tar xvf VOCdevkit_08-Jun-2007.tar
```

上述命令执行完毕后，应有以下结构。

```
$VOCdevkit/          # development kit
$VOCdevkit/VOCcode/   # VOC utility code
$VOCdevkit/VOC2007    # image sets, annotations, etc.
# ... and several other directories ...
```

(2) 为 PASCAL VOC 数据集创建 symlinks。

```
$ cd data
$ ln -s $VOCdevkit VOCdevkit2007
```

因为 PASCAL 数据集可能被多个项目使用，用 symlinks 会非常方便。

(3) 用相同的步骤获取 PASCAL VOC 2010 和 2012。

(4) 下载已训练 ImageNet models。

在论文中提到的已训练 ImageNet (包括 ZF, VGG_CNN_M_1024, VGG16) 模型可通过

执行下面命令获取:

```
$ ./data/scripts/fetch_imagenet_models.sh
```

也可以到 github 的 Caffe Model-Zoo 里获取 VGG 模型。链接如下:

<https://github.com/BVLC/caffe/wiki/Model-Zoo>

(5) 运行以下命令用 NIPS 2015 论文上的“alternating optimization”算法开始训练

```
$ ./experiments/scripts/faster_rcnn_alt_opt.sh
```

训练的输出在 output 目录下。

可根据实际情况在 faster_rcnn_alt_opt.sh 后加上参数, 命令形式和参数说明如下:

```
./experiments/scripts/faster_rcnn_alt_opt.sh [GPU_ID] [NET] [--set ...]
```

- GPU_ID 为训练所用的 GPU, 如果只有一块 GPU, 则为 0。
- NET 可选择 ZF, VGG_CNN_M_1024, VGG16。
- --set 允许配置 fast_rcnn.config 文件中的各种选项。

例如: --set EXP_DIR seed_rng1701 RNG_SEED 1701

(6) 运行以下命令对 Faster-RCNN 开始端对端训练, 训练的输出在 output 目录下。

```
$ ./experiments/scripts/faster_rcnn_end2end.sh
```

根据实际情况在 experiments/scripts/faster_rcnn_end2end.sh 后加上参数, 命令形式和参数说明如下:

```
./experiments/scripts/ experiments/scripts/faster_rcnn_end2end.sh [GPU_ID]  
[NET] [--set ...]
```

- GPU_ID 为训练所用的 GPU, 如果只有一块 GPU, 则为 0。
- NET 可选择 ZF, VGG_CNN_M_1024, VGG16。
- --set 允许配置 fast_rcnn.config 文件中的各种选项。

例如: `--set EXP_DIR seed_rng1701 RNG_SEED 1701`

这种训练方式把 RPN 和 Fast-RCNN 结合起来进行联合训练, 而不是把训练分成两个阶段, 训练的速度大约为分段训练的 1.5 倍, 训练所得模型的探测精度差不多。

(7) 运行以下命令开始测试已训练的模型, 测试结果输出到 `/output` 目录中。

```
./tools/test_net.py --gpu 1 --def models/VGG16/test.prototxt \
  --net output/default/voc_2007_trainval/vgg16_fast_rcnn_iter_40000.
caffemodel
```

2. Faster-RCNN VGG16模型的端对端训练的solver.prototxt解读

```
train_net: "models/pascal_voc/VGG16/faster_rcnn_end2end/train.prototxt" #
训练用的网络 prototxt 文件
base_lr: 0.001 #基本学习率
lr_policy: "step" #step 类型的 lr 策略, 具体的 step 由 stepsize 决定
gamma: 0.1
stepsize: 50000 #每 50000 次迭代时改变 base_lr
display: 20 #每 20 次迭代显示一次迭代信息
average_loss: 100
momentum: 0.9
# iter_size: 1
weight_decay: 0.0005
# We disable standard caffe solver snapshotting and implement our own snapshot
# function
#作者用自己开发的 snapshot 功能, 所以这里设为 0
snapshot: 0
# We still use the snapshot prefix, though
snapshot_prefix: "vgg16_faster_rcnn"
iter_size: 2 #iter_size*batch_size 次更新一次参数, iter_size 为 2 时, 可缩小
batch_size 为 1/2, 效果不变, 可有效减少内存的占用
```

3. Fast-rcnn caffe模型的train.prototxt解读

```
name: "VGG_ILSVRC_16_layers"
layer {
  name: 'input-data' #Python 层
  type: 'Python'
  top: 'data' #图像数据
  top: 'im_info' #图像信息
  top: 'gt_boxes' #Ground Truth 窗口
  python_param {
    module: 'roi_data_layer.layer' #Python 模块
    layer: 'RoIDataLayer' #RoIData 层
```

```

    param_str: "num_classes": 21" #一共 21 类
  }
}

layer {
  name: "conv1_1" #卷积层
  type: "Convolution"
  bottom: "data"
  top: "conv1_1"
  param {
    lr_mult: 0 #不参与微调学习
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  convolution_param {
    num_output: 64 #输出 64 个 channel
    pad: 1
    kernel_size: 3 #3x3 卷积核
  }
}

layer {
  name: "relu1_1" #ReLU 层
  type: "ReLU"
  bottom: "conv1_1"
  top: "conv1_1"
}

layer {
  name: "conv1_2" #卷积层
  type: "Convolution"
  bottom: "conv1_1"
  top: "conv1_2"
  param {
    lr_mult: 0 #不参与微调学习
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  convolution_param {
    num_output: 64 #输出 64 个 channel
    pad: 1
    kernel_size: 3 #3x3 卷积核
  }
}

layer {
  name: "relu1_2" #ReLU 层

```

```

    type: "ReLU"
    bottom: "conv1_2"
    top: "conv1_2"
}
layer {
    name: "pool1" #MAX Pooling 层
    type: "Pooling"
    bottom: "conv1_2"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
layer {
    name: "conv2_1" #卷积层
    type: "Convolution"
    bottom: "pool1"
    top: "conv2_1"
    param {
        lr_mult: 0 #不参与微调学习
        decay_mult: 0
    }
    param {
        lr_mult: 0
        decay_mult: 0
    }
    convolution_param {
        num_output: 128 #输出 128 个 channel
        pad: 1
        kernel_size: 3
    }
}
layer {
    name: "relu2_1" #ReLU 层
    type: "ReLU"
    bottom: "conv2_1"
    top: "conv2_1"
}
layer {
    name: "conv2_2" #卷积层
    type: "Convolution"
    bottom: "conv2_1"
    top: "conv2_2"
    param {
        lr_mult: 0 #不参与微调学习
        decay_mult: 0
    }
    param {

```

```

    lr_mult: 0
    decay_mult: 0
  }
  convolution_param {
    num_output: 128 #输出 128 个 channel
    pad: 1
    kernel_size: 3
  }
}
layer {
  name: "relu2_2" #ReLU 层
  type: "ReLU"
  bottom: "conv2_2"
  top: "conv2_2"
}
layer {
  name: "pool2" #MAX Pooling 层
  type: "Pooling"
  bottom: "conv2_2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv3_1" #卷积层
  type: "Convolution"
  bottom: "pool2"
  top: "conv3_1"
  param {
    lr_mult: 1 #从这一次卷积层开始 lr 不再是 0，开始学习，前面的卷积层固定，不进行微调
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 256 #输出 256 个 channel
    pad: 1
    kernel_size: 3 #3x3 卷积核
  }
}
layer {
  name: "relu3_1" #ReLU 层
  type: "ReLU"
  bottom: "conv3_1"
  top: "conv3_1"
}

```

学习


```

layer {
  name: "conv3_2" #卷积层
  type: "Convolution"
  bottom: "conv3_1"
  top: "conv3_2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 256 #输出 256 个 channel
    pad: 1
    kernel_size: 3
  }
}
layer {
  name: "relu3_2" #ReLU 层
  type: "ReLU"
  bottom: "conv3_2"
  top: "conv3_2"
}
layer {
  name: "conv3_3" #卷积层
  type: "Convolution"
  bottom: "conv3_2"
  top: "conv3_3"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 256 #输出 256 个 channel
    pad: 1
    kernel_size: 3 #3x3 卷积核
  }
}
layer {
  name: "relu3_3" #ReLU 层
  type: "ReLU"
  bottom: "conv3_3"
  top: "conv3_3"
}
layer {
  name: "pool3" #MAX Pooling 层
  type: "Pooling"
  bottom: "conv3_3"

```



```

    top: "pool3"
    pooling_param {
      pool: MAX
      kernel_size: 2
      stride: 2
    }
  }
  layer {
    name: "conv4_1" #卷积层
    type: "Convolution"
    bottom: "pool3"
    top: "conv4_1"
    param {
      lr_mult: 1
    }
    param {
      lr_mult: 2
    }
    convolution_param {
      num_output: 512 #输出 512 个 channel
      pad: 1
      kernel_size: 3 #3x3 卷积核
    }
  }
  layer {
    name: "relu4_1" #ReLU 层
    type: "ReLU"
    bottom: "conv4_1"
    top: "conv4_1"
  }
  layer {
    name: "conv4_2" #卷积层
    type: "Convolution"
    bottom: "conv4_1"
    top: "conv4_2"
    param {
      lr_mult: 1
    }
    param {
      lr_mult: 2
    }
    convolution_param {
      num_output: 512 #输出 512 个 channel
      pad: 1
      kernel_size: 3 #3x3 卷积核
    }
  }
  layer {
    name: "relu4_2" #ReLU 层
    type: "ReLU"

```

```

    bottom: "conv4_2"
    top: "conv4_2"
}
layer {
    name: "conv4_3" #卷积层
    type: "Convolution"
    bottom: "conv4_2"
    top: "conv4_3"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 512 #输出 512 个 channel
        pad: 1
        kernel_size: 3
    }
}
layer {
    name: "relu4_3" #ReLU 层
    type: "ReLU"
    bottom: "conv4_3"
    top: "conv4_3"
}
layer {
    name: "pool4" #MAX pooling 层
    type: "Pooling"
    bottom: "conv4_3"
    top: "pool4"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
layer {
    name: "conv5_1" #卷积层
    type: "Convolution"
    bottom: "pool4"
    top: "conv5_1"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 512 #输出 512 个 channel
        pad: 1

```

```

        kernel_size: 3
    }
}
layer {
    name: "relu5_1" #ReLU 层
    type: "ReLU"
    bottom: "conv5_1"
    top: "conv5_1"
}
layer {
    name: "conv5_2" #卷积层
    type: "Convolution"
    bottom: "conv5_1"
    top: "conv5_2"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 512 #输出 512 个卷积核
        pad: 1
        kernel_size: 3
    }
}
layer {
    name: "relu5_2" #ReLU 层
    type: "ReLU"
    bottom: "conv5_2"
    top: "conv5_2"
}
layer {
    name: "conv5_3" #卷积层
    type: "Convolution"
    bottom: "conv5_2"
    top: "conv5_3"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 512 #输出 512 个 channel
        pad: 1
        kernel_size: 3
    }
}
layer {

```



```

name: "relu5_3" #ReLU 层
type: "ReLU"
bottom: "conv5_3"
top: "conv5_3"
}

#===== RPN =====
#以下是 RPN 部分, RPN 部分是 Faster-RCNN 的精华所在, 用来产生建议窗口
layer {
  name: "rpn_conv/3x3" #卷积层
  type: "Convolution"
  bottom: "conv5_3"
  top: "rpn/output"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  convolution_param {
    num_output: 512 #输出 512 个 channel
    kernel_size: 3 pad: 1 stride: 1
    weight_filler { type: "gaussian" std: 0.01 } #从这一层开始不在 imagenet
    bias_filler { type: "constant" value: 0 }
  }
}
layer {
  name: "rpn_relu/3x3" #ReLU 层
  type: "ReLU"
  bottom: "rpn/output"
  top: "rpn/output"
}

layer {
  name: "rpn_cls_score" #卷积层
  type: "Convolution"
  bottom: "rpn/output"
  top: "rpn_cls_score"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  convolution_param {
    num_output: 18 # 2(bg/fg) * 9(anchors) #输出 18 个 channel, 对应背景/前景,
    kernel_size: 1 pad: 0 stride: 1
    weight_filler { type: "gaussian" std: 0.01 }
    bias_filler { type: "constant" value: 0 }
  }
}

layer {
  name: "rpn_bbox_pred" #卷积层
  type: "Convolution"
  bottom: "rpn/output"

```

```

    top: "rpn_bbox_pred"
    param { lr_mult: 1.0 }
    param { lr_mult: 2.0 }
    convolution_param {
        num_output: 36 # 4 * 9(anchors) #对应 9 个 anchors 的 location 的回归值
        <xcenter,ycenter,width,height>
        kernel_size: 1 pad: 0 stride: 1
        weight_filler { type: "gaussian" std: 0.01 }
        bias_filler { type: "constant" value: 0 }
    }
}

layer {
    bottom: "rpn_cls_score" #Reshape 层, reshape 方便 Softmax Loss 的计算
    top: "rpn_cls_score_reshape"
    name: "rpn_cls_score_reshape"
    type: "Reshape"
    reshape_param { shape { dim: 0 dim: 2 dim: -1 dim: 0 } }
}

layer {
    name: 'rpn-data'
    type: 'Python'
    bottom: 'rpn_cls_score'
    bottom: 'gt_boxes'
    bottom: 'im_info'
    bottom: 'data'
    top: 'rpn_labels' #Ground Truth Label (前景或背景), 用于分类的 Softmax Loss,
    整个 Python 层的处理复杂, 主要是通过 Anchor 和 Ground Truth Box 的 overlap 来决定是否是前
    景, 背景的使用是为了让网络得到更好的训练
    top: 'rpn_bbox_targets' #Bounding Box GroundTruth 回归值
    top: 'rpn_bbox_inside_weights' #Smooth L1 的 inside weight, 把 weight 分成
    inside 和 outside, 是 Faster-RCNN 的改进, Fast-RCNN 的 Smooth L1 是没有的
    top: 'rpn_bbox_outside_weights' #Smooth L1 的 outside weight
    python_param {
        module: 'rpn.anchor_target_layer'
        layer: 'AnchorTargetLayer'
        param_str: "'feat_stride': 16" #因为 feature map 的相邻两个特征点之间的距离
        相当于原图像 16 个 pixel 的距离, 所以这里用 16
    }
}

layer {
    name: "rpn_loss_cls" #RPN 的 Softmax Loss 层, 用于前景和背景归类回归
    type: "SoftmaxWithLoss"
    bottom: "rpn_cls_score_reshape"
    bottom: "rpn_labels"
    propagate_down: 1
    propagate_down: 0 #rpn_label 层属于数据层, 自然不需要 propagate down
    top: "rpn_cls_loss"

```



```

    loss_weight: 1 #Softmax Loss 在联合训练总 loss 中的权重
    loss_param {
        ignore_label: -1 #-1 是前景和背景之间的部分, 因为和前景和背景区别都不是足够大,
        所以忽略, 不参与 Softmax Loss 计算过程。
        normalize: true #归一化
    }
}

layer {
    name: "rpn_loss_bbox" #RPN Smooth L1 Loss 层, 用于 Bounding Box 回归值的回归
    type: "SmoothL1Loss"
    bottom: "rpn_bbox_pred" #Bounding Box 回归预测值, 注意这个值并不是 Bounding Box
    的实际坐标值, 而是 Anchor 产生的 Prior Box 到 Ground Truth Box 之间的回归值
    bottom: "rpn_bbox_targets" #Ground Truth 值
    bottom: 'rpn_bbox_inside_weights'
    bottom: 'rpn_bbox_outside_weights'
    top: "rpn_loss_bbox"
    loss_weight: 1 #Smooth L1 Loss 在联合训练总 loss 中的权重
    smooth_l1_loss_param { sigma: 3.0 }
}

#===== RoI Proposal =====

layer {
    name: "rpn_cls_prob"
    type: "Softmax"
    bottom: "rpn_cls_score_reshape"
    top: "rpn_cls_prob" #前景/背景分类概率
}

layer {
    name: 'rpn_cls_prob_reshape' #Reshape 层
    type: 'Reshape'
    bottom: 'rpn_cls_prob'
    top: 'rpn_cls_prob_reshape'
    reshape_param { shape { dim: 0 dim: 18 dim: -1 dim: 0 } }
}

layer {
    name: 'proposal' #Python 层, 上面说了 rpn_bbox_pred 只是回归值, 建议窗口 (RoI)
    必须是由 Anchor 产生的 Prior Box 经由 rpn_bbox_pred 调整后产生。
    type: 'Python'
    bottom: 'rpn_cls_prob_reshape'
    bottom: 'rpn_bbox_pred'
    bottom: 'im_info'
    top: 'rpn_rois'
    # top: 'rpn_scores'
    python_param {
        module: 'rpn.proposal_layer'
        layer: 'ProposalLayer'
    }
}

```

```

    param_str: "'feat_stride': 16"
  }
}

#layer {
#  name: 'debug-data'
#  type: 'Python'
#  bottom: 'data'
#  bottom: 'rpn_rois'
#  bottom: 'rpn_scores'
#  python_param {
#    module: 'rpn.debug_layer'
#    layer: 'RPNDebugLayer'
#  }
#}

layer {
  name: 'roi-data' #Python 层, 输出真正的建议窗口, 最终会限制在 300 个
  type: 'Python'
  bottom: 'rpn_rois'
  bottom: 'gt_boxes'
  top: 'rois'
  top: 'labels'
  top: 'bbox_targets'
  top: 'bbox_inside_weights'
  top: 'bbox_outside_weights'
  python_param {
    module: 'rpn.proposal_target_layer'
    layer: 'ProposalTargetLayer'
    param_str: "'num_classes': 21"
  }
}

#===== RCNN =====
#下面的部分和 Fast-RCNN 一致 (除了 Smooth L1 做了改进, 分 inside weights 和 outside
weights), 请参照 Fast-RCNN 相同的部分
layer {
  name: "roi_pool5"
  type: "ROIPooling"
  bottom: "conv5_3"
  bottom: "rois"
  top: "pool5"
  roi_pooling_param {
    pooled_w: 7
    pooled_h: 7
    spatial_scale: 0.0625 # 1/16
  }
}
layer {
  name: "fc6"
  type: "InnerProduct"

```

```

    bottom: "pool5"
    top: "fc6"
    param {
      lr_mult: 1
    }
    param {
      lr_mult: 2
    }
    inner_product_param {
      num_output: 4096
    }
  }
  layer {
    name: "relu6"
    type: "ReLU"
    bottom: "fc6"
    top: "fc6"
  }
  layer {
    name: "drop6"
    type: "Dropout"
    bottom: "fc6"
    top: "fc6"
    dropout_param {
      dropout_ratio: 0.5
    }
  }
  layer {
    name: "fc7"
    type: "InnerProduct"
    bottom: "fc6"
    top: "fc7"
    param {
      lr_mult: 1
    }
    param {
      lr_mult: 2
    }
    inner_product_param {
      num_output: 4096
    }
  }
  layer {
    name: "relu7"
    type: "ReLU"
    bottom: "fc7"
    top: "fc7"
  }
  layer {
    name: "drop7"
    type: "Dropout"
  }

```



```

    bottom: "fc7"
    top: "fc7"
    dropout_param {
      dropout_ratio: 0.5
    }
  }
  layer {
    name: "cls_score"
    type: "InnerProduct"
    bottom: "fc7"
    top: "cls_score"
    param {
      lr_mult: 1
    }
    param {
      lr_mult: 2
    }
    inner_product_param {
      num_output: 21
      weight_filler {
        type: "gaussian"
        std: 0.01
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
}
layer {
  name: "bbox_pred"
  type: "InnerProduct"
  bottom: "fc7"
  top: "bbox_pred"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 84
    weight_filler {
      type: "gaussian"
      std: 0.001
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}

```

```

}
layer {
  name: "loss_cls"
  type: "SoftmaxWithLoss"
  bottom: "cls_score"
  bottom: "labels"
  propagate_down: 1
  propagate_down: 0
  top: "loss_cls"
  loss_weight: 1
}
layer {
  name: "loss_bbox"
  type: "SmoothL1Loss"
  bottom: "bbox_pred"
  bottom: "bbox_targets"
  bottom: "bbox_inside_weights"
  bottom: "bbox_outside_weights"
  top: "loss_bbox"
  loss_weight: 1
}

```

4. 训练日志摘要

```

+ set -e
+ export PYTHONUNBUFFERED=True
+ PYTHONUNBUFFERED=True
+ GPU_ID=0
+ NET=VGG16
+ NET_lc=vgg16
+ DATASET=pascal_voc
+ array=($@)
+ len=3
+ EXTRA_ARGS=
+ EXTRA_ARGS_SLUG=
+ case $DATASET in
+ TRAIN_IMDB=voc_2007_trainval
+ TEST_IMDB=voc_2007_test
+ PT_DIR=pascal_voc
+ ITERS=70000
++ date +%Y-%m-%d_%H-%M-%S
+
LOG=experiments/logs/faster_rcnn_end2end_VGG16_.txt.2016-08-03_17-46-22
+ exec
++ tee -a experiments/logs/faster_rcnn_end2end_VGG16_.txt.2016-08-03-
17-46-22
+ echo Logging output to experiments/logs/faster_rcnn_end2end_VGG16_.txt.
2016-08-03_17-46-22
Logging output to experiments/logs/faster_rcnn_end2end_VGG16_.txt.2016-
08-03_17-46-22

```



```
+ ./tools/train_net.py --gpu 0 --solver models/pascal_voc/VGG16/faster_rcnn_end2end/solver.prototxt --weights data/imagenet_models/VGG16.v2.caffemodel --imdb voc_2007_trainval --iters 70000 --cfg experiments/cfgs/faster_rcnn_end2end.yml
```

Called with args:

```
Namespace(cfg_file='experiments/cfgs/faster_rcnn_end2end.yml', gpu_id=0, imdb_name='voc_2007_trainval', max_iters=70000, pretrained_model='data/imagenet_models/VGG16.v2.caffemodel', randomize=False, set_cfgs=None, solver='models/pascal_voc/VGG16/faster_rcnn_end2end/solver.prototxt')
```

Using config:

```
{'DATA_DIR': '/home/alex/workdir/py-faster-rcnn/data',
 'DEDUP_BOXES': 0.0625,
 'EPS': 1e-14,
 'EXP_DIR': 'faster_rcnn_end2end',
 'GPU_ID': 0,
 'MATLAB': 'matlab',
 'MODELS_DIR': '/home/alex/workdir/py-faster-rcnn/models/pascal_voc',
 'PIXEL_MEANS': array([[[ 102.9801, 115.9465, 122.7717]]]),
 'RNG_SEED': 3,
 'ROOT_DIR': '/home/alex/workdir/py-faster-rcnn',
 'TEST': {'BBOX_REG': True,
          'HAS_RPN': True,
          'MAX_SIZE': 1000,
          'NMS': 0.3,
          'PROPOSAL_METHOD': 'selective_search',
          'RPN_MIN_SIZE': 16,
          'RPN_NMS_THRESH': 0.7,
          'RPN_POST_NMS_TOP_N': 300,
          'RPN_PRE_NMS_TOP_N': 6000,
          'SCALES': [600],
          'SVM': False},
 'TRAIN': {'ASPECT_GROUPING': True,
           'BATCH_SIZE': 128,
           'BBOX_INSIDE_WEIGHTS': [1.0, 1.0, 1.0, 1.0],
           'BBOX_NORMALIZE_MEANS': [0.0, 0.0, 0.0, 0.0],
           'BBOX_NORMALIZE_STDS': [0.1, 0.1, 0.2, 0.2],
           'BBOX_NORMALIZE_TARGETS': True,
           'BBOX_NORMALIZE_TARGETS_PRECOMPUTED': True,
           'BBOX_REG': True,
           'BBOX_THRESH': 0.5,
           'BG_THRESH_HI': 0.5,
           'BG_THRESH_LO': 0.0,
           'FG_FRACTION': 0.25,
           'FG_THRESH': 0.5,
           'HAS_RPN': True,
           'IMS_PER_BATCH': 1,
           'MAX_SIZE': 1000,
           'PROPOSAL_METHOD': 'gt',
           'RPN_BATCHSIZE': 256,
           'RPN_BBOX_INSIDE_WEIGHTS': [1.0, 1.0, 1.0, 1.0],
           'RPN_CLOBBER_POSITIVES': False,
```

```

'RPN_FG_FRACTION': 0.5,
'RPN_MIN_SIZE': 16,
'RPN_NEGATIVE_OVERLAP': 0.3,
'RPN_NMS_THRESH': 0.7,
'RPN_POSITIVE_OVERLAP': 0.7,
'RPN_POSITIVE_WEIGHT': -1.0,
'RPN_POST_NMS_TOP_N': 2000,
'RPN_PRE_NMS_TOP_N': 12000,
'SCALES': [600],
'SNAPSHOT_INFIX': '',
'SNAPSHOT_ITERS': 10000,
'USE_FLIPPED': True,
'USE_PREFETCH': False},
'USE_GPU_NMS': True}
Loaded dataset `voc_2007_trainval` for training
Set proposal method: gt
Appending horizontally-flipped training examples...
voc_2007_trainval gt roidb loaded from /home/alex/workdir/py-faster-rcnn/
data/cache/voc_2007_trainval_gt_roidb.pkl
done
Preparing training data...
done
.....
.....
.....
AP for aeroplane = 0.6941
AP for bicycle = 0.7935
AP for bird = 0.6695
AP for boat = 0.5665
AP for bottle = 0.4870
AP for bus = 0.7790
AP for car = 0.8006
AP for cat = 0.7920
AP for chair = 0.4858
AP for cow = 0.7478
AP for diningtable = 0.6523
AP for dog = 0.8142
AP for horse = 0.8077
AP for motorbike = 0.7542
AP for person = 0.7719
AP for pottedplant = 0.4188
AP for sheep = 0.6854
AP for sofa = 0.6479
AP for train = 0.7569
AP for tvmonitor = 0.7151
Mean AP = 0.6920
~~~~~
Results:
0.694
0.794
0.670

```

```

0.566
0.487
0.779
0.801
0.792
0.486
0.748
0.652
0.814
0.808
0.754
0.772
0.419
0.685
0.648
0.757
0.715
0.692
.....

```

```

-----
Results computed with the **unofficial** Python eval code.
Results should be very close to the official MATLAB eval code.
Recompute with `./tools/reval.py --matlab ...` for your paper.
-- Thanks, The Management
-----

```

```

real    12m28.626s
user    10m20.416s
sys     2m13.284s

```

参考文献

- [1] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In CVPR, 2014.
- [2] R. Girshick. Fast R-CNN. arXiv:1504.08083, 2015.
- [3] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders. Selective search for object recognition. IJCV, 2013.
- [4] Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards real-time object detection with region proposal networks. In: NIPS. (2015).
- [5] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.

15

第 15 章 SSD 模型

在前面章节中，我们介绍了 R-CNN^[1]系列模型，尤其是 R-CNN 系列的最新成果 Faster-RCNN^[2]模型一定给大家留下了深刻印象。但是，我们自然会进一步考虑一个问题，Faster-RCNN 还能进一步改进吗？Faster-RCNN 处理很快，但是能够在实时系统中应用吗？答案是否定的。Faster-RCNN 的性能相当不错，但无法应用于实时系统。这是因为在实时系统中，视频每秒达到 30 帧，Faster-RCNN 在 NVIDIA Titan X 上，即使用最小的 ZF 模型也只能达到每秒实时处理 17 帧（mAP 62.1），而采用 VGG16^[3]模型则只能达到每秒实时处理 7 帧（mAP 73.2）。值得庆幸的是，最近目标探测和实时性方法的研究又有了重大突破，突破的方法就是本章要介绍的内容——SSD^[4]模型。

15.1 SSD 模型简介

SSD 模型（Single Shot Multibox Detector）是为了解决实时探测目标定位的问题，它的探测精度很高，使用 VGG16 模型，mAP 达到 72.1，接近 Faster-RCNN 的精度，探测速度更快，在 NVIDIA Titan X 上，采用 cuDNN V5，处理速度达到每秒 72 帧。

SSD 采用的是端对端的训练方法。和 Faster-RCNN 的 Anchor 不同之处在于，SSD 在多个 feature map 上进行处理，因为每一层 feature map 的感受也不同。Faster RCNN 是先提

取 Proposal Box (建议框), 然后再分类, 而 SSD 则利用 Anchor 直接进行分类和 Bounding Box 回归。总的来说, SSD 继承了 Yolo 模型的特点, 并结合了 Faster-RCNN 中采用 anchor 的方法。

SSD 模型网络的结构图如图 15-1 所示, 并对比了它和 Yolo^[5]模型的区别。

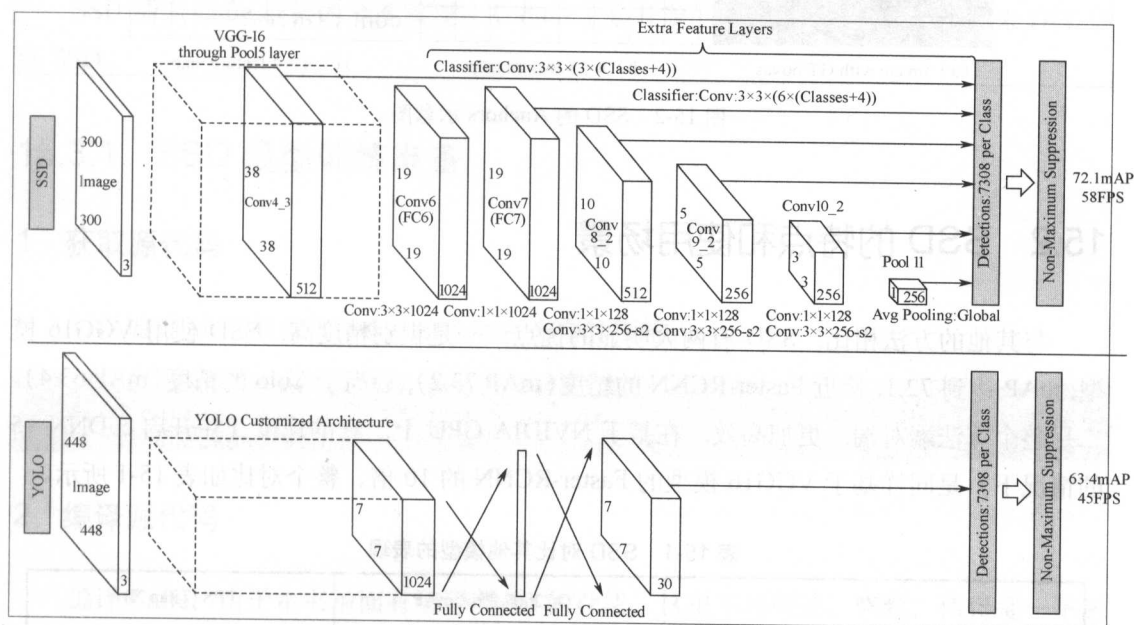


图 15-1 SSD 和 YOLO 模型结构图

SSD 只需要在训练时直接输入图像、图像标签和 Ground Truth Bounding Box (图 15-2 (a) 子图两个动物所在的框)。在 SSD 网络中, 我们在多个 feature map 的每个特征点上应用不同 Anchor 生成 Prior Box 来预测物体的分类和 Bounding Box 回归值。

图 15-2 (b) 子图和 (c) 子图中的虚线框都为 Anchor, (b) 子图代表 4 种 anchor, 8 个 Prior Box。

SSD 模型具体内容较为冗长, 这里不再一一叙述, 有兴趣的读者可阅读论文原文“SSD: Single Shot MultiBox Detector”。

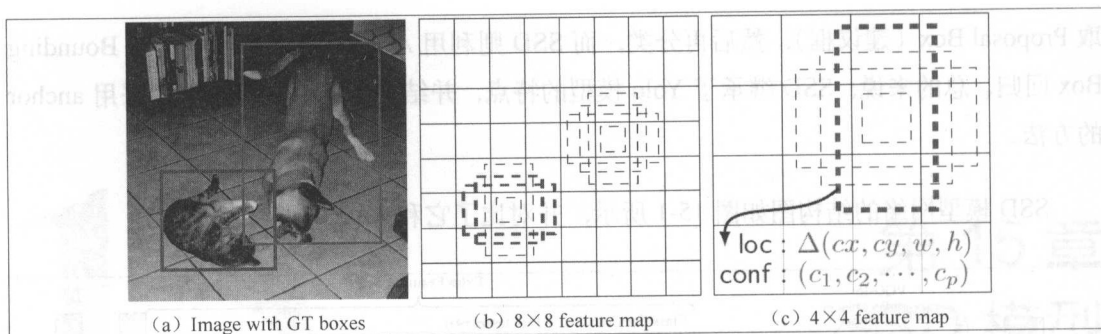


图 15-2 SSD 的 Anchors 示意图

15.2 SSD 的特点和使用场景

与其他的方法相比, SSD 有两大明显的优点: 一是识别精度高, SSD 使用 VGG16 模型, mAP 达到 72.1, 接近 Faster-RCNN 的精度 (mAP 73.2), 远高于 Yolo 的精度 (mAP 63.4)。二是整个算法端对端, 更加高效, 在基于 NVIDIA GPU 上, 它的速度 (在开启 cuDNN v5 的情况下) 是同样基于 VGG16 模型的 Faster-RCNN 的 10 倍。整个对比如表 15-1 所示。

表 15-1 SSD 对比其他模型的表现

模型	VOC2007 测试 mAP	FPS (Titan X)
Faster-RCNN (VGG 16)	73.2	7
Faster-RCNN (ZF)	62.1	17
Yolo	63.4	45
Fast Yolo	52.7	155
SSD300 (VGG16)	72.1	58
SSD300 (VGG16, cuDNN v5)	72.1	72
SSD500 (VGG16)	75.1	23

SSD300 指的是输入图片尺寸为 300×300 。SSD500 指的是输入图片尺寸为 500×500 。

由于 SSD 精确度高, 速度快, 已能满足实时系统需要, 相信在实时系统探测识别定位的使用场景中非常有价值。当然如果在嵌入式系统中, 由于缺乏 Titan X 这样高效的 GPU,

还是无法达到实时处理，但至少让业界看到了希望。

15.3 Caffe SSD 解读

SSD 基于 Caffe 框架的全部实现在 Github 上是开源的。我们可以按以下步骤来 DIY 训练 SSD。

15.3.1 SSD 模型训练准备

1. 获取源代码

可以通过 git 工具直接下载，在 Linux 命令行下输入：

```
$ git clone https://github.com/weiliu89/caffe.git
$ cd caffe
$ git checkout ssd
```

2. 编译源代码

编译 Caffe，由于本书前面有编译 Caffe 的介绍，这里不再赘述。当然，值得提一下的是，由于 CPU 运行实在太慢了，所以 SSD 的设计是完全基于 GPU 的，如果读者要自己训练 SSD，必须至少使用一块 NVIDIA 的 GPU。另外这块 GPU 必须有足够大的显存。在笔者的电脑上，训练需要略小于 8GB 显存。这意味着 4GB 或 6GB 显存的 GPU 无法直接训练 SSD，当然可以通过减小 batchsize 来试着绕过这个问题。

3. 下载预训练模型

下载 VGG16 fully convolutional reduced (atrous) Net 预训练 caffemodel。下载链接如下：<https://gist.github.com/weiliu89/2ed6e13bfd5b57cf81d6>。

注意：由于国内网络限制原因，可能造成无法访问，请读者自行解决。

4. 下载VOC2007和VOC2012数据集

请读者自行访问相关网站下载数据集,并用 `tar -xvf` 来解包到相应目录。SSD 默认 VOC 数据集目录下 `ssd/caffe/data` 下。

```
$ wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCTrainval_11-May-2012.tar
$ wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCTrainval_06-Nov_2007.tar
$ wget
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtest_06-Nov-2007.tar
$ tar -xvf VOCTrainval_11-May-2012.tar
$ tar -xvf VOCTrainval_06-Nov_2007.tar
$ tar -xvf VOCtest_06-Nov-2007.tar
```

5. 创建LMDB数据库

在 `ssd/caffe` 目录下执行以下命令:

```
$ ./data/VOC0712/create_list.sh
$ ./data/VOC0712/create_dats.sh
```

创建完后,应该有下列文件产生。

```
./data/VOCdevkit/VOC0712/lmdb/VOC0712_trainval_lmdb
./data/VOCdevkit/VOC0712/lmdb/VOC0712_test_lmdb
```

15.3.2 SSD 模型训练

1. 训练过程

在 `Caffe` 目录下执行以下命令开始训练:

```
$ python examples/ssd/ssd_pascal.py
```

第一次运行,一般会出现错误,原因可能是 GPU 的设置问题。源代码中的 GPU 设置默认是 4 个 GPU 并行训练,但一般情况下在一台电脑上只有一个 GPU,所以需要修改 `ssd_pascal.py` 文件。

打开 `examples/ssd` 下的 `ssd_pascal.py`, 搜索 `gpus = "0,1,2,3"`,并修改为 `gpus = "0"`,代表当

前的电脑只有 1 个 GPU。

另外，一些当前的运行环境可能缺少某些 Python 库，读者可根据错误提示通过 pip install 自行安装。

完成上述排错后，再运行 `python examples/ssd/ssd_pascal.py`，训练应该正常开始。

下面我们对 SSD 模型的 `solver.prototxt` 和 `train.prototxt` 进行详细分析，并给出中文注解。这里值得提一下的是，新接触 SSD 的读者可能对找不到 `train.prototxt` 和 `test.prototxt` 相应的文件，原因是 SSD 的模型非常大，整个完整的 `train.prototxt` 长达 1600 多行，一旦某些参数需要改动，修改的工作量很大。而且，`train.prototxt` 一旦改动，`test.prototxt` 和 `deploy.prototxt` 也必须做相应修改。所以 SSD 的原作者采用了一个非常有效的方法：用“`ssd_pascal.py`” Python 脚本自动生成 `solver.prototxt`，`train.prototxt`，`test.prototxt` 和 `deploy.prototxt`。

2. 测试过程

SSD 提供了 Python 类型的测试代码，我们可以在训练完毕后，在 Caffe 目录下直接运行以下命令：

```
$ python examples/ssd/score_ssd_pascal.py
```

测试运行根据 GPU 的不同，可能需要运行一段时间，此时屏幕上没有任何输出，静待一会儿，就会看到测试结果。

3. solver.prototxt 解读

```
train_net: "models/VGGNet/VOC0712/SSD_300x300/train.prototxt" #训练用的网络
prototxt 文件
test_net: "models/VGGNet/VOC0712/SSD_300x300/test.prototxt" #测试用的网络
prototxt 文件
test_iter: 4952 #测试时的迭代次数
test_interval: 10000 #每 10000 次迭代测试一次
base_lr: 0.001 #基本学习率
display: 10 #每 10 次迭代显示一次迭代信息
max_iter: 60000
lr_policy: "step" #step 类型的 lr 策略，具体的 step 由 stepsize 决定
gamma: 0.1
```

```

momentum: 0.9
weight_decay: 0.0005
stepsize: 40000 #每 40000 次迭代时改变 base_lr, 由于最大迭代次数为 60000 次, 所以改
变 base_lr 只有一次, 即在 40000 次迭代时, base_lr 从 0.001 变为 0.0001
snapshot: 40000 #每 40000 次迭代抓一次快照
snapshot_prefix:
"models/VGGNet/VOC0712/SSD_300x300/VGG_VOC0712_SSD_300x300"
solver_mode: GPU #GPU 方式
device_id: 0 #GPU 0
debug_info: false #关闭 debug 输出
snapshot_after_train: true #训练完抓一次 snapshot
test_initialization: false #不做测试初始化, 如果为 True, 在迭代次数为 0 时执行一次
测试。如果在 solver 中不写 test_initialization, 默认是 True
average_loss: 10 #每 10 次迭代平均一下 loss, 以防止 loss 的跳变
iter_size: 1 #读取 batchsize*itersize 个图像才做一下 gradient decent, 显存不够
的读者可以把 batchsize 减半, 同时把 itersize 翻倍来降低显存的开销。不过训练速度会减慢。
type: "SGD" #随机梯度下降
eval_type: "detection" #测试类型为 "detection", caffe 测试类型分
"classification"和"detection", 目标检测定位用"detection", 如果是 Alexnet 等分类模型,
用"classification", 如果在 solver 中不写 eval_type, 默认是 classification。
ap_version: "11point" #计算 AP (Average Precision) 的方法, VOC2007 采用 11point
方式计算 AP。caffe 另外还支持 MaxIntegral 和 Integral 方式。

```

train.prototxt 解读

由于训练网络过大, 我们挑选部分要点讲解。

```

name: "VGG_VOC0712_SSD_300x300_train"
layer {
  name: "data"
  type: "AnnotatedData" #Annotated Data 类型的数据层, 由 SSD 的原作者实现
  top: "data" #top[0] blob 为图像
  top: "label" # top[1] blob 为标签 (包括 Ground Truth Bounding Box)
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true #支持图像镜像
    mean_value: 104 #Blue 层的 mean value
    mean_value: 117 #Green 层的 mean value
    mean_value: 123 #Red 层的 mean value
    resize_param {
      prob: 1 #采用这种 resize 策略的概率, 这边因为只有一种 resize 策略, 所以概率为 1
      resize_mode: WARP #拉伸模式: 当图像的尺寸不是 300x300 时, 拉伸图像到 300x300
      height: 300 #最后用于训练的图像尺寸 300x300
      width: 300
      interp_mode: LINEAR #放大图像时需要内插像素, 内插像素有很多模式, 这里定义的是
几个方式, 随机选取。内插模式和 OpenCV 中定义的一致
      interp_mode: AREA

```



```

    interp_mode: NEAREST
    interp_mode: CUBIC
    interp_mode: LANCZOS4
  }
  emit_constraint {
    emit_type: CENTER #用 Ground Truth Bounding Box 的中心坐标是否落在 crop box
里作为这个 crop box 是否应注释的依据。比如 Ground Truth Bounding Box 类别为“飞机”,中心
坐标落在 crop box 中,则标记 crop box 的类别也为“飞机”
  }
}
data_param {
  source: "examples/VOC0712/VOC0712_trainval_lmdb" #训练用的数据库
  batch_size: 8 #每次迭代用的 batch 中有 8 幅图像
  backend: LMDB #LMDB 类型数据源
}
annotated_data_param { #这里定义了一组 batch_sampler,在训练时,此层会随机选用
sampler 对 batch 的图像进行处理,用这种方法可以增加训练数据的随机性和多样性
  batch_sampler { #原图,不做任何处理
    max_sample: 1
    max_trials: 1 #采用的最大尝试次数
  }
  batch_sampler { #缩放因子为 0.3~1.0 之间的随机值,缩放长宽比为 0.5~2.0 之间的随
机值
    sampler {
      min_scale: 0.3
      max_scale: 1.0
      min_aspect_ratio: 0.5
      max_aspect_ratio: 2.0
    }
    sample_constraint {
      min_jaccard_overlap: 0.1 #用 minimum jaccard overlap 作为判断是否注解
为 positive 标签的一个条件
    }
    max_sample: 1
    max_trials: 50
  }
  batch_sampler {
    sampler {
      min_scale: 0.3
      max_scale: 1.0
      min_aspect_ratio: 0.5
      max_aspect_ratio: 2.0
    }
    sample_constraint {
      min_jaccard_overlap: 0.3
    }
    max_sample: 1
    max_trials: 50
  }
}

```

```

batch_sampler {
  sampler {
    min_scale: 0.3
    max_scale: 1.0
    min_aspect_ratio: 0.5
    max_aspect_ratio: 2.0
  }
  sample_constraint {
    min_jaccard_overlap: 0.5
  }
  max_sample: 1
  max_trials: 50
}
batch_sampler {
  sampler {
    min_scale: 0.3
    max_scale: 1.0
    min_aspect_ratio: 0.5
    max_aspect_ratio: 2.0
  }
  sample_constraint {
    min_jaccard_overlap: 0.7
  }
  max_sample: 1
  max_trials: 50
}
batch_sampler {
  sampler {
    min_scale: 0.3
    max_scale: 1.0
    min_aspect_ratio: 0.5
    max_aspect_ratio: 2.0
  }
  sample_constraint {
    min_jaccard_overlap: 0.9
  }
  max_sample: 1
  max_trials: 50
}
batch_sampler {
  sampler {
    min_scale: 0.3
    max_scale: 1.0
    min_aspect_ratio: 0.5
    max_aspect_ratio: 2.0
  }
  sample_constraint {
    max_jaccard_overlap: 1.0
  }
  max_sample: 1
  max_trials: 50
}

```

```

    }
    label_map_file: "data/VOC0712/labelmap_voc.prototxt" #label map file
    路径和文件名
  }
}

```

#VGG16 网络的卷积部分本文直接跳过，重点讲解 SSD 特有的部分。

#VGG16 网络的卷积部分从 conv1_1 layer 到 pool5 layer。

#下面从 fc6，即全卷积网络开始讲解，这里的全卷积网络替代 VGG16 的全连接网络。

```

layer {
  name: "fc6" #fc6 为卷积层
  type: "Convolution"
  bottom: "pool5"
  top: "fc6"
  param {
    lr_mult: 1 #weights 的 lr 乘数
    decay_mult: 1
  }
  param {
    lr_mult: 2 #bias 的 lr 乘数
    decay_mult: 0
  }
  convolution_param {
    num_output: 1024 #对应 GCC16 的第一个 1024 个神经元的全连接层
    pad: 6 #每边加 6 个像素的 pad
    kernel_size: 3 #卷积核为 3x3
    weight_filler {
      type: "xavier" #xavier 类型的初始化
    }
    bias_filler {
      type: "constant" #bias 的初始化为常量 0
      value: 0
    }
    dilation: 6 #用来扩大卷积核，扩大后用 0 填充因为扩大引起的洞
  }
}
layer {
  name: "relu6" #relu6 为 ReLU 层
  type: "ReLU"
  bottom: "fc6"
  top: "fc6"
}
layer {
  name: "fc7" #卷积层
  type: "Convolution"
  bottom: "fc6"
  top: "fc7"
  param {
    lr_mult: 1
    decay_mult: 1
  }
}

```



```

    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    convolution_param {
      num_output: 1024 #对应 GCC16 的第二个 1024 个神经元的全连接层
      kernel_size: 1 #用 1x1 的卷积核来表示两个全连接层相连
      weight_filler {
        type: "xavier"
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
}
layer {
  name: "relu7" #ReLU 层
  type: "ReLU"
  bottom: "fc7"
  top: "fc7"
}
layer {
  name: "conv6_1" #卷积层
  type: "Convolution"
  bottom: "fc7"
  top: "conv6_1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256 #256 个 channel
    pad: 0
    kernel_size: 1
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
}
layer {

```

```

    name: "conv6_1_relu" #ReLU 层
    type: "ReLU"
    bottom: "conv6_1"
    top: "conv6_1"
}
layer {
    name: "conv6_2" #卷积层
    type: "Convolution"
    bottom: "conv6_1"
    top: "conv6_2"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 512 #512 个 channel
        pad: 1
        kernel_size: 3
        stride: 2
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}
}
layer {
    name: "conv6_2_relu" #ReLU 层
    type: "ReLU"
    bottom: "conv6_2"
    top: "conv6_2"
}
layer {
    name: "conv7_1" #卷积层
    type: "Convolution"
    bottom: "conv6_2"
    top: "conv7_1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
}

```



```

convolution_param {
  num_output: 128 #128 个 channel
  pad: 0
  kernel_size: 1
  stride: 1
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
}
layer {
  name: "conv7_1_relu" #ReLU 层
  type: "ReLU"
  bottom: "conv7_1"
  top: "conv7_1"
}
layer {
  name: "conv7_2" #卷积层
  type: "Convolution"
  bottom: "conv7_1"
  top: "conv7_2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
}
convolution_param {
  num_output: 256 #256 个 channel
  pad: 1
  kernel_size: 3
  stride: 2
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
}
layer {
  name: "conv7_2_relu" #ReLU 层
  type: "ReLU"
  bottom: "conv7_2"

```

```

    top: "conv7_2"
  }
  layer {
    name: "conv8_1" #卷积层
    type: "Convolution"
    bottom: "conv7_2"
    top: "conv8_1"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    convolution_param {
      num_output: 128 #128 个 channel
      pad: 0
      kernel_size: 1
      stride: 1
      weight_filler {
        type: "xavier"
      }
      bias_filler {
        type: "constant"
        value: 0
      }
    }
  }
  layer {
    name: "conv8_1_relu" #ReLU 层
    type: "ReLU"
    bottom: "conv8_1"
    top: "conv8_1"
  }
  layer {
    name: "conv8_2" #卷积层
    type: "Convolution"
    bottom: "conv8_1"
    top: "conv8_2"
    param {
      lr_mult: 1
      decay_mult: 1
    }
    param {
      lr_mult: 2
      decay_mult: 0
    }
    convolution_param {
      num_output: 256 #256 个 channel
      pad: 1

```

```

    kernel_size: 3
    stride: 2
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "conv8_2_relu" #ReLU 层
  type: "ReLU"
  bottom: "conv8_2"
  top: "conv8_2"
}
layer {
  name: "pool6" #AVE Pooling 层
  type: "Pooling"
  bottom: "conv8_2"
  top: "pool6"
  pooling_param {
    pool: AVE
    global_pooling: true #整个 feature map 做平均 pooling
  }
}

```

#下面的 Normalize 层是 SSD 对 caffe 层的扩展, 是把 feature map 进行 Normalize 处理, 在 across_spatial 为 false 的情况下, Normalization 后的每个点 \hat{x} 按以下公式处理。

$$\hat{x} = \frac{x}{\|x\|_2}$$

$$\text{其中 } \|x\|_2 = \left(\sum_{i=1}^d |x_i|^2 \right)^{\frac{1}{2}}$$

其中 \hat{x} 为 Normalization 后的 feature map 上的点, x 为 Normalization 前的 feature map 上对应位置的点。SSD 的原作者在其另一篇论文《ParseNet: Looking Wider to See Better》详细介绍了这种方法。由于 conv6_2, conv7_2, conv8_2, pool6 和 fc7 都会做这样的处理, 所以注解是相同的, 所以这里不再重复注解。

```

layer {
  name: "conv4_3_norm"
  type: "Normalize"
  bottom: "conv4_3"
  top: "conv4_3_norm"
  norm_param {
    across_spatial: false
    scale_filler {
      type: "constant"
      value: 20
    }
  }
  channel_shared: false
}

```



```

}
}

```

conv4_3_norm_mbox_loc 这个卷积层用来预测回归值, 输出 channel 为 12, 为 Bounding Box 回归值的预测值, 预测值共有 12 个。12 代表 3 anchors * 4。这里 anchor 就是一个预定义的矩形框, 3 个 anchors 长宽比基于 1:1, 1:2, 2:1, 每个 anchors 需要 4 个回归值 <x center, y center, width, height>, 分别对中心坐标和 Bounding Box 的 width 和 height 做回归。这个预测值最后会和由 Ground Truth 值做 Smooth L1 回归。Smooth L1 回归据笔者所知, 是在 Fast-RCNN 中首次提出的, 具体见 Fast-RCNN 论文中所述。值的注意是: 这里原作者认为所有的类别的 Bounding Box 回归值是共享的。即不管此处识别是“飞机”还是“电视机”, 都共享一个回归值。在 SSD 的源代码中, 也可以通过配置 share_location=False 来让每一类拥有自己的回归值, 因为一共有 21 类物体 (包括背景), 这时输出 channel 为变为 $3 \times 4 \times 21 = 252$ 。但也许是由于代码未完成或其他原因, 笔者试着训练了一下, 发现无法收敛。

```

layer {
  name: "conv4_3_norm_mbox_loc"
  type: "Convolution"
  bottom: "conv4_3_norm"
  top: "conv4_3_norm_mbox_loc"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 12
    pad: 1
    kernel_size: 3 #3x3 的卷积核表示我们识别的是 feature map 上 3x3 的区域
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "conv4_3_norm_mbox_loc_perm"
  #Permute层, 当前 shape 为<Num of Pictures, Channels, Width, Height>, 但为了后面计算 Loss 方便, 所以把 shape 变更为<Num of Pictures, Width, Height, Channels>, 这样用 Flatten 和 Concatenate 就能把所有层的 Bounding Box 预测回归值的 blob 合成一个大的 blob, 很巧妙的方法!
  type: "Permute"
  bottom: "conv4_3_norm_mbox_loc"
  top: "conv4_3_norm_mbox_loc_perm"
}

```

```

    permute_param {
      order: 0
      order: 2
      order: 3
      order: 1
    }
  }
}

```

#conv4_3_norm_mbox_loc_flat 是 Flatten 层, 把 Width, Height, Channels 三个 axis 的数据平铺

```

layer {
  name: "conv4_3_norm_mbox_loc_flat"
  type: "Flatten"
  bottom: "conv4_3_norm_mbox_loc_perm"
  top: "conv4_3_norm_mbox_loc_flat"
  flatten_param {
    axis: 1 #表示从 axis 开始平铺数据
  }
}

```

#conv4_3_norm_mbox_conf 这个卷积层用来预测 feature map 上每个点的类别, 输出 channel 为 63, 这里解释一下为什么是 63? $63=3*21$, 因为我们一共采用了三个 anchors, 3 个 anchors 长宽比分别基于 1: 1, 1:2, 2:1, 21 为一共识别 21 类, 其中 20 类为物体, 1 类为背景。这样的话, 我们对每个 anchor 都做了 21 类的识别

```

layer {
  name: "conv4_3_norm_mbox_conf"
  type: "Convolution"
  bottom: "conv4_3_norm"
  top: "conv4_3_norm_mbox_conf"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 63
    pad: 1
    kernel_size: 3 #3x3 的卷积核表示我们识别的是 feature map 上 3x3 的区域
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
}

```

conv4_3_norm_mbox_conf_perm 是 Permute 层, 当前 shape 为<Num of Pictures,

Channels, Width, Height>, 但为了后面计算 Loss 方便, 所以通过 Permute 层把 shape 变更为 <Num of Pictures, Width, Height, Channels>, 这样用 Flatten 和 Concatenate 就能把所有层的分类识别值的 blob 合成一个大的 blob

```
layer {
  name: "conv4_3_norm_mbox_conf_perm"
  type: "Permute"
  bottom: "conv4_3_norm_mbox_conf"
  top: "conv4_3_norm_mbox_conf_perm"
  permute_param {
    order: 0
    order: 2
    order: 3
    order: 1
  }
}
```

#Flatten 层, 把 Width, Height, Channels 三个 axis 的数据平铺

```
layer {
  name: "conv4_3_norm_mbox_conf_flat"
  type: "Flatten"
  bottom: "conv4_3_norm_mbox_conf_perm"
  top: "conv4_3_norm_mbox_conf_flat"
  flatten_param {
    axis: 1
  }
}
```

#Prior Box 产生层, 在训练时, Proposal Box 是必需的, 这里的 Prior Box 就是 Proposal Box, SSD 算法要求我们在每个做识别的卷积层上用滑动窗口的模式生成 Prior Box, 且使用不同类型的 Anchor, 假设 conv4_3 这个 feature map 长宽都为 N, 则我们一共有 $N \times N$ 的特征点, 对每个特征点套用三个 Anchors, 则我们一共有 $N \times N \times 3$ 个 Prior Box。由于我们知道训练图像中所有 Ground Truth Bounding Box 的大小和位置, 所以这些 Prior Box 和 Ground Truth 窗口进行计算, 就能得到 Ground Truth 的 Bounding Box 的回归值, 在最后的 loss 层中, Ground Truth 的 Bounding Box 回归值最终会和 conv4_3_norm_mbox_loc 里生成的 Bounding Box 回归预测值做 Smooth L1 Loss。通过大量数据的训练, 最终使得 Loss 最小化, Bounding Box 回归预测值就接近于 Ground Truth 的 Bounding Box 的回归值。这时, 我们就可以认为整个网络有了直接预测 Bounding Box 回归值的功能。

在识别定位过程中, 我们输入一张待识别定位图片进入整个网络, 会得到 Prior Box 和 Bounding Box 回归预测值, 然后把 Prior Box 通过 Bounding Box 回归预测值校正一下, 最后就得到的 Bounding Box 的预测值。假设某个 Prior Box - P 的值为 $\langle P_x, P_y, P_w, P_h \rangle$, 通过网络生成的 Bounding Box 回归预测值为 $\langle dx(P), dy(P), dw(P), dh(P) \rangle$, 最终的 Bounding Box 的预测值 $\langle G_x, G_y, G_w, G_h \rangle$ 由下面公式产生:

$$\begin{aligned} G_x &= P_x dx(P) + P_x \\ G_y &= P_y dy(P) + P_y \\ G_w &= P_w \exp(dw(P)) \\ G_h &= P_h \exp(dh(P)) \end{aligned}$$

注: x, y 为 Bounding Box 的中心点坐标, w, h 为 Bounding Box 的宽和高。

```
layer {
  name: "conv4_3_norm_mbox_priorbox"
```

```

type: "PriorBox"
  bottom: "conv4_3_norm"
  bottom: "data"
  top: "conv4_3_norm_mbox_priorbox"
  prior_box_param {
    min_size: 30.0
    aspect_ratio: 2 #因为 flip 为 true, 所以我们会 有 1:2 和 2:1
    flip: true
    clip: true #让归一化的 Prior 坐标控制在 [0,1] 之间, 截掉 [0,1] 之外的部分
    variance: 0.1 #回归值的原始值太小, 用 variance 来放大原始值, 0.1 为 10 倍
    variance: 0.1
    variance: 0.2
    variance: 0.2
  }
}

```

fc7_mbox_loc 这个卷积层用来预测回归值, 输出 channel 为 24, 为 Bounding Box 回归值的预测值, 预测值共有 24 个。12 代表 6 anchors * 4。这里 anchor 就是一个预定义的矩形框, 6 个 anchors 如下:

- 长宽比 1:1 (基于 fc7_mbox_priorbox 的 min_size)
- 长宽比 1:1 (基于 fc7_mbox_priorbox 的 min_size 和 max_size 的几何平均值)
- 长宽比 1:2 (基于 fc7_mbox_priorbox 的 min_size)
- 长宽比 2:1 (基于 fc7_mbox_priorbox 的 min_size)
- 长宽比 1:3 (基于 fc7_mbox_priorbox 的 min_size)
- 长宽比 3:1 (基于 fc7_mbox_priorbox 的 min_size)

每个 anchors 需要 4 个回归值 $\langle x \text{ center}, y \text{ center}, \text{width}, \text{height} \rangle$, 分别对中心坐标和 Bounding Box 的 width 和 height 做回归。这个预测值最后会和由 Ground Truth 值做 Smooth L1 回归。Smooth L1 回归据笔者所知, 是在 Fast-RCNN 中首次提出的, 具体见 Fast-RCNN 论文中所述。值的注意是: 这里原作者认为所有的类别的 Bounding Box 回归值是共享的。即不管此处识别是“飞机”还是“电视机”, 都共享一个回归值。在 SSD 的源代码中, 也可以通过配置 share_location=False 来让每一类拥有自己的回归值, 因为一共有 21 类物体(包括背景), 这时输出 channel 变为 $6*4*21=504$ 。

```

layer {
  name: "fc7_mbox_loc"
  type: "Convolution"
  bottom: "fc7"
  top: "fc7_mbox_loc"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 24
    pad: 1
    kernel_size: 3
    stride: 1
  }
}

```

```

weight_filler {
  type: "xavier"
}
bias_filler {
  type: "constant"
  value: 0
}
}
}

```

#Permute 层, 当前 shape 为<Num of Pictures, Channels, Width, Height>, 但为了后面计算 Loss 方便, 所以把 shape 变更为<Num of Pictures, Width, Height, Channels >, 这样用 Flatten 和 Concatenate 就能把所有层的 Bounding Box 预测回归值的 blob 合成一个大的 blob

```

layer {
  name: "fc7_mbox_loc_perm"
  type: "Permute"
  bottom: "fc7_mbox_loc"
  top: "fc7_mbox_loc_perm"
  permute_param {
    order: 0
    order: 2
    order: 3
    order: 1
  }
}

```

#fc7_mbox_loc_flat 是 Flatten 层, 把 Width, Height, Channels 三个 axis 的数据铺开

```

layer {
  name: "fc7_mbox_loc_flat"
  type: "Flatten"
  bottom: "fc7_mbox_loc_perm"
  top: "fc7_mbox_loc_flat"
  flatten_param {
    axis: 1
  }
}

```

#conv4_3_norm_mbox_conf 这个卷积层用来预测 feature map 上每个点的类别, 输出 channel 为 126, 这里解释一下为什么是 126? $126=6*21$, 因为我们一共采用了 6 个 anchors, 6 个 anchors 的定义见前面的注解, 21 为一共识别 21 类, 其中 20 类为物体, 1 类为背景。这样的话, 我们对每个 anchor 都做了 21 类的识别。

```

layer {
  name: "fc7_mbox_conf"
  type: "Convolution"
  bottom: "fc7"
  top: "fc7_mbox_conf"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {

```



```

        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 126
        pad: 1
        kernel_size: 3
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}

```

#Permute 层, 当前 shape 为<Num of Pictures, Channels, Width, Height>, 但为了后面计算 Loss 方便, 所以把 shape 变更为<Num of Pictures, Width, Height, Channels >, 这样用 Flatten 和 Concatenate 就能把所有层的 Bounding Box 预测回归值的 blob 合成一个大的 blob。

```

layer {
    name: "fc7_mbox_conf_perm"
    type: "Permute"
    bottom: "fc7_mbox_conf"
    top: "fc7_mbox_conf_perm"
    permute_param {
        order: 0
        order: 2
        order: 3
        order: 1
    }
}

```

#fc7_mbox_conf_flat 是 Flatten 层, 把 Width, Height, Channels 三个 axis 的数据平铺。

```

layer {
    name: "fc7_mbox_conf_flat"
    type: "Flatten"
    bottom: "fc7_mbox_conf_perm"
    top: "fc7_mbox_conf_flat"
    flatten_param {
        axis: 1
    }
}

```

#Prior Box 产生层, 在训练时, Ground Truth 是必须的, 这里的 Prior Box 就是 Ground Truth, SSD 算法要求我们在每个做识别的卷积层上用滑动窗口的模式生成 Prior Box, 且使用不同类型的 Anchor, 假设 fc7 这个 feature map 长宽都为 N, 则我们一共有 $N*N$ 的特征点, 对每个特征点套用三个 Anchors, 则我们一共有 $N*N*6$ 个 Prior Box。

```

layer {
    name: "fc7_mbox_priorbox"
    type: "PriorBox"
    bottom: "fc7"
}

```

```

bottom: "data"
top: "fc7_mbox_priorbox"
prior_box_param {
  min_size: 60.0
  max_size: 114.0
  aspect_ratio: 2
  aspect_ratio: 3
  flip: true
  clip: true
  variance: 0.1
  variance: 0.1
  variance: 0.2
  variance: 0.2
}
}

#####
#由于 conv6_2, conv7_2, conv8_2, pool6 和 fc7 的注解是相同的, 所以这里不再重复注解
#下面的注解直接从 mbox_loc 开始。
#####

# mbox_loc 把 conv4_3, fc7, conv6_2, conv7_2, conv8_2, pool6 的 Bounding Box
回归预测值输出 blob 合成一个大的 blob, 便于后面的 Smooth L1 Loss 处理。
layer {
  name: "mbox_loc"
  type: "Concat"
  bottom: "conv4_3_norm_mbox_loc_flat"
  bottom: "fc7_mbox_loc_flat"
  bottom: "conv6_2_mbox_loc_flat"
  bottom: "conv7_2_mbox_loc_flat"
  bottom: "conv8_2_mbox_loc_flat"
  bottom: "pool6_mbox_loc_flat"
  top: "mbox_loc"
  concat_param {
    axis: 1
  }
}

# mbox_conf 把 conv4_3, fc7, conv6_2, conv7_2, conv8_2, pool6 的分类预测值输出
blob 合成一个大的 blob, 便于后面的 Softmax Loss 处理。
layer {
  name: "mbox_conf"
  type: "Concat"
  bottom: "conv4_3_norm_mbox_conf_flat"
  bottom: "fc7_mbox_conf_flat"
  bottom: "conv6_2_mbox_conf_flat"
  bottom: "conv7_2_mbox_conf_flat"
  bottom: "conv8_2_mbox_conf_flat"
  bottom: "pool6_mbox_conf_flat"
  top: "mbox_conf"
  concat_param {
    axis: 1
  }
}

```



```

    }
}
# mbox_priorbox 把 conv4_3、fc7、conv6_2、conv7_2、conv8_2、pool6 的 Prior Box
输出 blob 合成一个大的 blob，便于后面的 Smooth L1 Loss 和 Softmax Loss 处理。
layer {
    name: "mbox_priorbox"
    type: "Concat"
    bottom: "conv4_3_norm_mbox_priorbox"
    bottom: "fc7_mbox_priorbox"
    bottom: "conv6_2_mbox_priorbox"
    bottom: "conv7_2_mbox_priorbox"
    bottom: "conv8_2_mbox_priorbox"
    bottom: "pool6_mbox_priorbox"
    top: "mbox_priorbox"
    concat_param {
        axis: 2 # 为什么是 2? 因为 axis 0 是图像数，axis 1 是 2 个 channels，一个 channel
        存放 Prior Box 坐标的 mean，另一个 channel 存放 Prior Box 坐标的 variance
    }
}
# mbox_loss 层属于 MultiBoxLoss 类型，这种类型的层是 SSD 的原作者自定义的，是在 caffe
层上的扩展。总的来说 MultiBoxLoss 类型 loss 层内嵌了 2 个 loss 层，一个是 Smooth L1 Loss
层，用于 Bounding Box 回归。另一个是 Softmax Loss 层，用于分类回归。由于篇幅原因，我们这
里给个总体的注解。
总的 loss = loc_weight * SMOOTH_L1_loss + SOFTMAX_loss
SMOOTH_L1_loss 通过 mbox_loc、mbox_priorbox、label 这 3 个 blob 来计算 loss。
SOFTMAX_loss 通过 mbox_conf、label 这 2 个 blob 来计算 loss。
layer {
    name: "mbox_loss"
    type: "MultiBoxLoss"
    bottom: "mbox_loc"
    bottom: "mbox_conf"
    bottom: "mbox_priorbox"
    bottom: "label"
    top: "mbox_loss"
    include {
        phase: TRAIN
    }
    propagate_down: true #bottom[0] blob 有 backward
    propagate_down: true #bottom[1] blob 有 backward
    propagate_down: false #bottom[2] blob 没有 backward
    propagate_down: false #bottom[3] blob 没有 backward
    loss_param {
        normalization: VALID
    }
    multibox_loss_param {
        loc_loss_type: SMOOTH_L1 # Bounding Box 回归采用 SMOOTH_L1 回归
        conf_loss_type: SOFTMAX # 类别回归采用 SOFTMAX 回归
        loc_weight: 1.0 # SMOOTH_L1 loss 前的权重
        num_classes: 21 # 一共 21 类
    }
}

```

```

share_location: true #前面提到的所有类共享 Bounding Box 回归值
match_type: PER_PREDICTION
overlap_threshold: 0.5 #Prior Box 和 Ground Truth Bounding Box 的 overlap
超过门限为有效的 positive 类别
use_prior_for_matching: true
background_label_id: 0 #背景类别 id 为 0
use_difficult_gt: true #使用困难模式的 Ground Truth 标签
do_neg_mining: true #为 true 时, 会挖掘背景进行训练
neg_pos_ratio: 3.0 #背景和 positive 类别的比例, 非背景的类型都是 positive 类别
neg_overlap: 0.5 #Prior Box 和 Ground Truth Bounding Box 的 overlap 低于门
限为有效的 neg_overlap 类别
code_type: CENTER_SIZE #从回归值生成 Bounding Box 的方法
}
}

```

4. 训练日志摘要

```

./build/tools/caffe: /home/alex/anaconda2/lib/libtiff.so.5: no version
information available (required by /usr/local/lib/libopencv_highgui.so.2.4)
./build/tools/caffe: /home/alex/anaconda2/lib/liblzma.so.5: no version
information available (required by /usr/lib/x86_64-linux-gnu/libunwind.so.8)
./build/tools/caffe: /home/alex/anaconda2/lib/liblzma.so.5: no version
information available (required by /usr/lib/x86_64-linux-gnu/libavcodec-ffmpeg.
so.56)
I0713 20:52:03.513489 1661 caffe.cpp:185] Using GPUs 0
I0713 20:52:03.985041 1661 caffe.cpp:190] GPU 0: GeForce GTX 1070
I0713 20:52:04.156756 1661 solver.cpp:51] Initializing solver from
parameters:
train_net: "models/VGGNet/VOC0712/SSD_300x300/train.prototxt"
test_net: "models/VGGNet/VOC0712/SSD_300x300/test.prototxt"
test_iter: 4952
test_interval: 10000
base_lr: 0.001
display: 10
max_iter: 60000
lr_policy: "step"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0005
stepsize: 40000
snapshot: 40000
snapshot_prefix:
"models/VGGNet/VOC0712/SSD_300x300/VGG_VOC0712_SSD_300x300"
solver_mode: GPU
device_id: 0
debug_info: false
snapshot_after_train: true
test_initialization: false
average_loss: 10

```

```

iter_size: 1
type: "SGD"
eval_type: "detection"
ap_version: "l1point"
I0713 20:52:04.156852 1661 solver.cpp:84] Creating training net from
train_net file: models/VGGNet/VOC0712/SSD_300x300/train.prototxt
I0713 20:52:04.157873 1661 net.cpp:49] Initializing net from parameters:
name: "VGG_VOC0712_SSD_300x300_train"
state {
  phase: TRAIN
}
.....
.....
.....
I0713 20:52:04.349189 1661 layer_factory.hpp:77] Creating layer data
I0713 20:52:04.349236 1661 net.cpp:91] Creating Layer data
I0713 20:52:04.349243 1661 net.cpp:399] data -> data
I0713 20:52:04.349253 1661 net.cpp:399] data -> label
I0713 20:52:04.349982 1673 db_lmdb.cpp:35] Opened lmdb examples/VOC0712/
VOC0712_test_lmdb
I0713 20:52:04.353070 1661 annotated_data_layer.cpp:52] output data size:
1,3,300,300
I0713 20:52:04.355473 1661 net.cpp:141] Setting up data
I0713 20:52:04.355497 1661 net.cpp:148] Top shape: 1 3 300 300 (270000)
I0713 20:52:04.355502 1661 net.cpp:148] Top shape: 1 1 2 8 (16)
I0713 20:52:04.355505 1661 net.cpp:156] Memory required for data: 1080064
I0713 20:52:04.355510 1661 layer_factory.hpp:77] Creating layer data_data_
0_split
I0713 20:52:04.355525 1661 net.cpp:91] Creating Layer data_data_0_split
.....
.....
.....
I0713 20:52:04.624706 1661 caffe.cpp:219] Starting Optimization
I0713 20:52:04.624728 1661 solver.cpp:282] Solving VGG_VOC0712_SSD_
300x300_train
I0713 20:52:04.624732 1661 solver.cpp:283] Learning Rate Policy: step
I0713 20:52:07.678120 1661 solver.cpp:231] Iteration 20, loss = 13.1966
I0713 20:52:07.678150 1661 solver.cpp:247] Train net output #0:
mbox_loss = 11.9325 (* 1 = 11.9325 loss)
I0713 20:52:07.678174 1661 sgd_solver.cpp:106] Iteration 20, lr = 0.001
I0713 20:53:03.634416 1661 solver.cpp:231] Iteration 30, loss = 12.4803
I0713 20:53:03.634486 1661 solver.cpp:247] Train net output #0:
mbox_loss = 12.2206 (* 1 = 12.2206 loss)
I0713 20:53:03.634493 1661 sgd_solver.cpp:106] Iteration 30, lr = 0.001
I0713 20:53:23.453970 1661 solver.cpp:231] Iteration 40, loss = 12.0543
I0713 20:53:23.454000 1661 solver.cpp:247] Train net output #0:
mbox_loss = 12.0289 (* 1 = 12.0289 loss)
I0713 20:53:23.454006 1661 sgd_solver.cpp:106] Iteration 40, lr = 0.001
I0713 20:53:41.742501 1661 solver.cpp:231] Iteration 50, loss = 10.0162
I0713 20:53:41.742583 1661 solver.cpp:247] Train net output #0:
mbox_loss = 9.31762 (* 1 = 9.31762 loss)

```

```

.....
.....
I0715 03:49:08.408432 1661 solver.cpp:231] Iteration 59970, loss = 1.76305
I0715 03:49:08.408458 1661 solver.cpp:247] Train net output #0:
mbox_loss = 2.00899 (* 1 = 2.00899 loss)
I0715 03:49:08.408463 1661 sgd_solver.cpp:106] Iteration 59970, lr = 0.0001
I0715 03:49:26.934960 1661 solver.cpp:231] Iteration 59980, loss = 1.95824
I0715 03:49:26.935034 1661 solver.cpp:247] Train net output #0:
mbox_loss = 1.82409 (* 1 = 1.82409 loss)
I0715 03:49:26.935040 1661 sgd_solver.cpp:106] Iteration 59980, lr = 0.0001
I0715 03:49:45.398707 1661 solver.cpp:231] Iteration 59990, loss = 1.72689
I0715 03:49:45.398736 1661 solver.cpp:247] Train net output #0:
mbox_loss = 1.90606 (* 1 = 1.90606 loss)
I0715 03:49:45.398741 1661 sgd_solver.cpp:106] Iteration 59990, lr = 0.0001
I0715 03:50:02.096882 1661 solver.cpp:581] Snapshotting to binary proto file
models/VGGNet/VOC0712/SSD_300x300/VGG_VOC0712_SSD_300x300_iter_60000.caffemodel
I0715 03:50:02.964320 1661 sgd_solver.cpp:273] Snapshotting solver state
to binary proto file models/VGGNet/VOC0712/SSD_300x300/VGG_VOC0712_SSD_
300x300_iter_60000.solverstate
I0715 03:50:03.860791 1661 solver.cpp:320] Iteration 60000, loss = 1.87887
I0715 03:50:03.860817 1661 solver.cpp:421] Iteration 60000, Testing net
(#0)
I0715 03:50:03.860848 1661 net.cpp:684] Ignoring source layer mbox_loss
I0715 03:51:59.156328 1661 solver.cpp:531] Test net output #0:
detection_eval = 0.726887
I0715 03:51:59.156427 1661 solver.cpp:325] Optimization Done.
I0715 03:51:59.156431 1661 caffe.cpp:222] Optimization Done.

```

参考文献

- [1] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In CVPR, 2014.
- [2] Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards real-time object detection with region proposal networks. In: NIPS. (2015).
- [3] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.
- [4] W. Liu¹, D. Anguelov², D. Erhan³, C. Szegedy³, S. Reed⁴, C. Fu¹, A. C. Berg, SSD: Single Shot MultiBox Detector, arXiv preprint arXiv:1512.02325, 2015.
- [5] Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: CVPR. (2016).

16

第 16 章 Kaggle 项目实践： 人脸特征检测

Kaggle 是全球著名的数据分析竞赛网站。在 Kaggle 网站上，有两类项目：一类是真正的用于竞赛的项目，在此类项目的竞赛中排名靠前的选手可以分享项目奖金。另外一类是知识类项目，用于知识的学习和技能的训练。知识类项目对初入数据分析领域的读者非常有实践意义的。本章我们将讲解如果用 Caffe 架构解决 Kaggle 的一个知识类项目：检测人脸图像特征点。

16.1 项目简介

检测人脸图像特征点是 Kaggle 的一个知识类项目。项目的目的是预测人脸图像的关键点位置。这个项目在实际工程中具有积极意义，人脸图像特征点是人脸的关键特征。项目在很多实际场景得到了应用，比较典型应用有：

- 图像和视频中的人脸跟踪
- 面部表情的分析

- 检测畸形的面部标志的医疗诊断
- 生物特征识别/人脸识别

面部特征点检测是一个非常具有挑战性的问题。面部特征从一个人到另一个人有很大的不同，甚至一个单一的个人，也会由于三维姿态、大小、位置、视角和照明等条件产生很大的变化。计算机视觉的研究为了解决这些困难，已经走了很长的路，但仍然有许多改进的机会。

在 Kaggle 网站上提供了一个基准数据集和一个 R 语言环境下的教程，帮助初入数据分析领域的人员学习如何去分析人脸图像。本章将介绍在 Python 环境下，如何运用 Caffe 框架，使用深度学习来实践这个项目。以下为这个项目的链接 URL，有兴趣的读者可自行访问。

```
https://www.kaggle.com/c/facial-keypoints-detection
```

友情提醒：读者在国内注册 Kaggle 账号时，会出现无法验证通过的情况。主要原因是 Kaggle 使用了 Google 的服务，请读者自行解决。一旦账号注册成功，后续就可以正常使用。

16.2 赛题和数据

项目提供以下数据文件进行训练和测试。

training.csv：用以训练的 7049 张人脸图像。采用 CSV 格式，每一行包含 15 个关键特征点坐标和图像数据的像素。

test.csv：用以测试的 1783 张测试图像。采用 CSV 格式，每一行包含 imageid 和图像数据的像素行。

一共有 15 个关键特征点需要预测。这 15 个关键点如下所示：

- left_eye_center

- right_eye_center
- left_eye_inner_corner
- left_eye_outer_corner
- right_eye_inner_corner
- right_eye_outer_corner
- left_eyebrow_inner_end
- left_eyebrow_outer_end
- right_eyebrow_inner_end
- right_eyebrow_outer_end
- nose_tip
- mouth_left_corner
- mouth_right_corner
- mouth_center_top_lip
- mouth_center_bottom_lip

在一些数据样本中, 标签有缺失的情况, 所以我们在数据预处理时要把这些数据处理掉。

输入图像是在数据文件的最后一个字段中给出的, 由一个像素的列表组成(按行排序), 像素取值范围为 0 ~ 255 的 Integer。图像维度是 96 × 96 像素, 只有一个通道(黑白图片), 如图 16-1 所示。

数据下载链接 URL 如下:

```
https://www.kaggle.com/c/facial-keypoints-detection/data
```

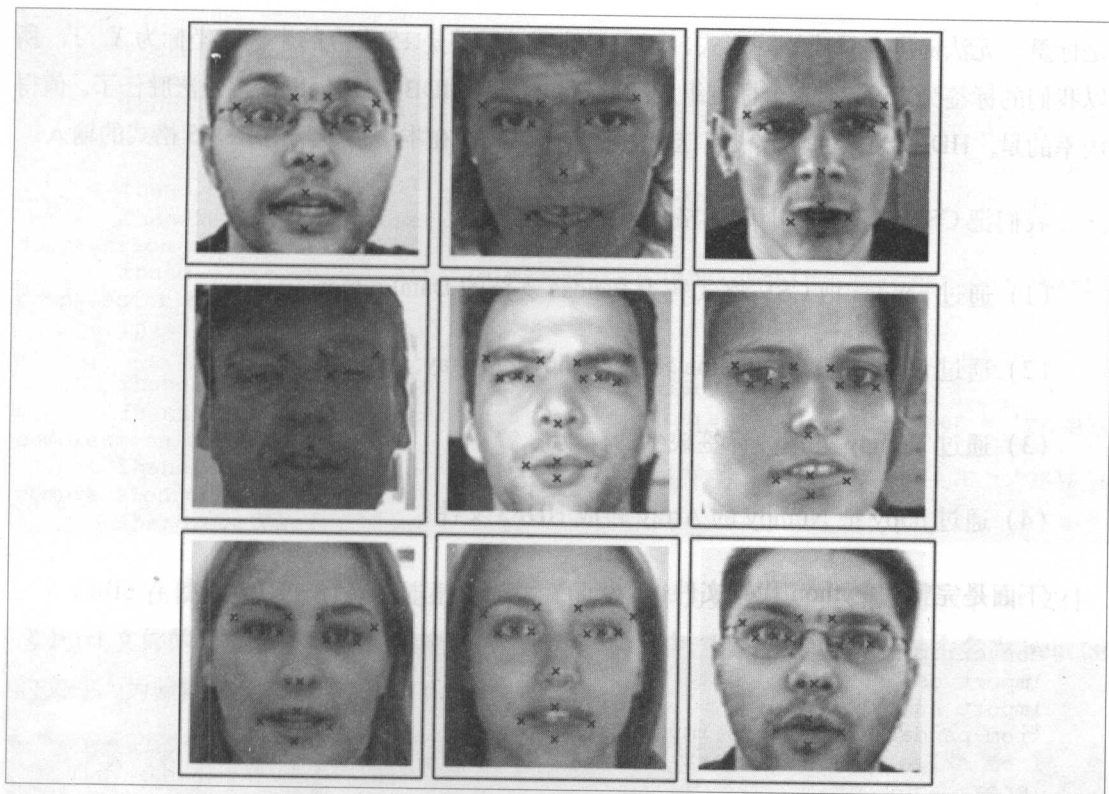


图 16-1 人脸特征点示意图

16.3 Caffe 训练和测试数据库

16.3.1 数据库生成

Kaggle 网站上的项目往往提供 CSV 格式的原始数据, CSV 格式的数据在 Python 语言或 R 语言中处理非常方便。但 Caffe 自身并不支持 CSV 格式, 只支持 LevelDB、LMDB、HDF5 这三种格式。这就意味着 CSV 格式的原始数据无法直接作为数据层在 Caffe 中进行训练。

在本项目中, 项目数据为 CSV 格式, 所以必须先把 CSV 格式转成 Caffe 支持的格式。一般来说, LMDB 格式是最高效的数据库格式, 也是 Caffe 中最常用的数据库格式, 但 Caffe 支持的 LMDB 或 LevelDB 格式有一个局限性, 即 LMDB 或 LevelDB 中提供训练的标签只能

是标量, 无法提供向量或矩阵形式。而我们的项目标签是 15 个点的坐标。坐标为 X, Y , 所以我们的标签为 $15 \times 2 = 30$ 个标量组成的向量。这样 LMDB 或 LevelDB 就无法胜任了。值得庆幸的是, HDF5 格式的输入没有这个限制。所以我们在本项目中采用 HDF5 格式的输入。

我们把 CSV 文件转成 HDF5 格式采用如下步骤:

- (1) 通过 pandas 把 CSV 格式转为 Pandas 支持的 DataFrame;
- (2) 通过 pandas 把 DataFrame 转成 Numpy 的 Array;
- (3) 通过 Numpy 处理掉标签缺少的样本;
- (4) 通过 h5py 把 Numpy 的 Array 转成 HDF5 文件。

下面是完整的 Python 代码文件。

```
convert.py
import os
import numpy as np
from pandas.io.parsers import read_csv
from sklearn.utils import shuffle
import h5py

TRAIN_CSV = '/to/your/path/training.csv'

def csv_to_hd5():
    dataframe = read_csv(os.path.expanduser(TRAIN_CSV))
    dataframe['Image'] = dataframe['Image'].apply(lambda img: np.fromstring
(img, sep = ' '))
    dataframe = dataframe.dropna() #把缺失数据的图像丢弃掉
    data = np.vstack(dataframe['Image'].values) / 255.

    label = dataframe[dataframe.columns[:-1]].values
    label = (label - 48) / 48
    data, label = shuffle(data, label, random_state = 0) #随机化

    return data, label

if __name__ == '__main__':
    # train_data/val_data
    data, label = csv_to_hd5()
    data = data.reshape(-1, 1, 96, 96) #图像为 channel: 1 height: 96 width: 96
    data_train = data[:-100, :, :, :] #把最后 100 张图像作为验证图像
    data_val = data[-100:, :, :, :]
```

```

# train_label/val_data
label = label.reshape(-1,1,1,30)
label_train = label[:-100,:,:,:]
label_val = label[-100,:,:,:]

fhandle = h5py.File('train.hd5','w') #train 数据库
fhandle.create_dataset('data', data = data_train, compression = 'gzip',
compression_opts = 4)
fhandle.create_dataset('label', data = label_train, compression = 'gzip',
compression_opts = 4)
fhandle.close()

fhandle = h5py.File('val.hd5','w') #validation 数据库
fhandle.create_dataset('data', data = data_val, compression = 'gzip',
compression_opts = 4)
fhandle.create_dataset('label', data = label_val, compression = 'gzip',
compression_opts = 4)
fhandle.close()

```

Caffe 在使用 HDF5 时, 不能把 .hd5 格式的文件作为数据源, 必须采用一个 txt 文件, 这个 txt 文件的内容为 .hd5 文件的全路径文件名。本例中, 我们为训练建立一个名为 train.txt 的文件, 为验证建立一个名为 val.txt 的文件。

```

train.txt
/to/your/path/train.hd5
val.txt
/to/your/path/val.hd5

```

16.3.2 网络对比

在本项目的实现中, 笔者提供了两种训练网络, 这两种训练网络都能实现脸部特征点的检测, 但精确度有所区别:

网络一: 只有两个全连接层的网络。

网络二: 有若干卷积层的网络 (此网络由笔者从 Caffe 自带的 LeNet 改动而来)。

从最后的训练结果看, 网络二的最终 Loss 远远小于网络一的最终 Loss。我们可以得出网络二优于网络一。

16.3.3 网络一

结构图如图 16-2 所示。

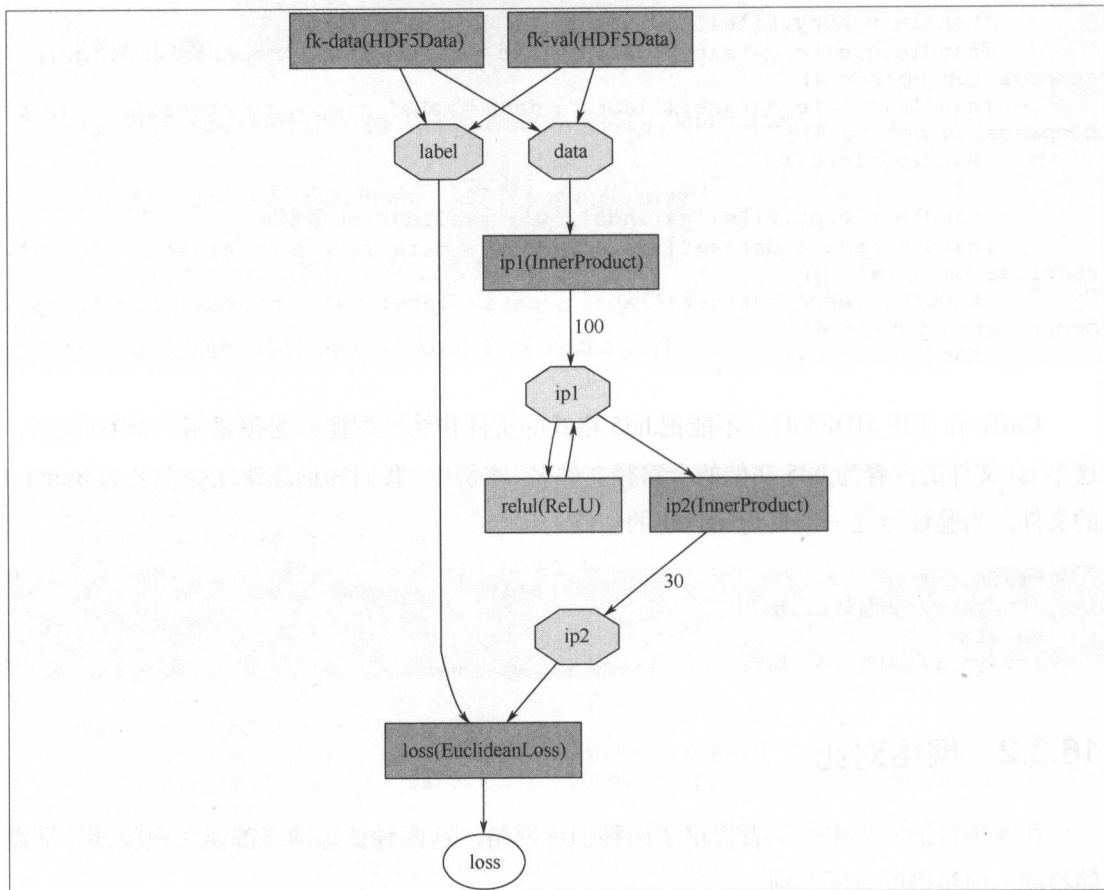


图 16-2 网络一的结构图

1. fk_solver.prototxt解读

```

# The train/test net protocol buffer definition
net: "/to/your/path/fk_train_val.prototxt"
test_iter: 1
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.001 #基本学习率
momentum: 0.9

```

```

weight_decay: 0.0005
# The learning rate policy
lr_policy: "fixed"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000 #最大10000 次迭代
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "/to/your/path/fk"
# solver mode: CPU or GPU
solver_mode: GPU

```

2. fk_train_val.prototxt解读

```

name: "FK"
layer {
  name: "fk-data"
  type: "HDF5Data" #HDF5 格式
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  hdf5_data_param {
    source: "/to/your/path/train.txt"
    batch_size: 64
  }
}
layer {
  name: "fk-val"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  hdf5_data_param {
    source: "/to/your/path/val.txt"
    batch_size: 100
  }
}
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip1"
}

```

```

param {
  lr_mult: 1
}
param {
  lr_mult: 2
}
inner_product_param {
  num_output: 100
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 30 #这里的30代码15种特征的坐标<X,Y>
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
}
layer {
  name: "loss"
  type: "EuclideanLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
}

```

3. 训练过程摘要

```

./build/tools/caffe train --solver=/to/your/path/fk_solver.prototxt

I0921 10:21:08.887480 19254 caffe.cpp:185] Using GPUs 0
I0921 10:21:08.895900 19254 caffe.cpp:190] GPU 0: GeForce GTX 1070
I0921 10:21:09.349082 19254 solver.cpp:48] Initializing solver from
parameters:
  test_iter: 100
  test_interval: 500
  base_lr: 0.01
  display: 100
  max_iter: 10000
  lr_policy: "fixed"
  gamma: 0.0001
  power: 0.75
  momentum: 0.9
  weight_decay: 0.0005
  snapshot: 5000
  snapshot_prefix: "/to/your/path/fk"
  solver_mode: GPU
  device_id: 0
  net: "/to/your/path/fk_train_val.prototxt"
I0921 10:21:09.349191 19254 solver.cpp:91] Creating training net from net
file: /to/your/path/fk_train_val.prototxt
I0921 10:21:09.349292 19254 net.cpp:313] The NetState phase (0) differed from
the phase (1) specified by a rule in layer fk-val
I0921 10:21:09.349331 19254 net.cpp:49] Initializing net from parameters:
  name: "FK"
  .....
I0921 10:28:48.999646 19254 solver.cpp:228] Iteration 9000, loss = 0.0824755
I0921 10:28:48.999670 19254 solver.cpp:244]   Train net output #0: loss =
0.0823925 (* 1 = 0.0823925 loss)
I0921 10:28:48.999675 19254 sgd_solver.cpp:106] Iteration 9000, lr=0.00617924
I0921 10:29:01.656446 19254 solver.cpp:228] Iteration 9100, loss = 0.0734249
I0921 10:29:01.656472 19254 solver.cpp:244]   Train net output #0: loss =
0.0733419 (* 1 = 0.0733419 loss)
I0921 10:29:01.656477 19254 sgd_solver.cpp:106] Iteration 9100, lr=0.00615496
I0921 10:29:14.202775 19254 solver.cpp:228] Iteration 9200, loss = 0.0614149
I0921 10:29:14.202801 19254 solver.cpp:244]   Train net output #0: loss =
0.061332 (* 1 = 0.061332 loss)
I0921 10:29:14.202811 19254 sgd_solver.cpp:106] Iteration 9200, lr=0.0061309
I0921 10:29:26.852851 19254 solver.cpp:228] Iteration 9300, loss = 0.0656728
I0921 10:29:26.852903 19254 solver.cpp:244]   Train net output #0: loss =
0.0655899 (* 1 = 0.0655899 loss)
I0921 10:29:26.852910 19254 sgd_solver.cpp:106] Iteration 9300, lr =
0.00610706
I0921 10:29:36.887555 19254 solver.cpp:228] Iteration 9400, loss = 0.0636958
I0921 10:29:36.887590 19254 solver.cpp:244]   Train net output #0: loss =
0.0636129 (* 1 = 0.0636129 loss)
I0921 10:29:36.887596 19254 sgd_solver.cpp:106] Iteration 9400, lr =

```



```

0.00608343
I0921 10:29:37.039644 19254 solver.cpp:337] Iteration 9500, Testing net (#0)
I0921 10:29:37.039674 19254 net.cpp:684] Ignoring source layer fk-data
I0921 10:29:37.197420 19254 solver.cpp:404] Test net output #0: loss =
0.0650894 (* 1 = 0.0650894 loss)
I0921 10:29:37.198459 19254 solver.cpp:228] Iteration 9500, loss = 0.0579248
I0921 10:29:37.198475 19254 solver.cpp:244] Train net output #0: loss =
0.0578418 (* 1 = 0.0578418 loss)
I0921 10:29:37.198483 19254 sgd_solver.cpp:106] Iteration 9500, lr =
0.00606002
I0921 10:29:37.327309 19254 solver.cpp:228] Iteration 9600, loss = 0.064568
I0921 10:29:37.327339 19254 solver.cpp:244] Train net output #0: loss =
0.064485 (* 1 = 0.064485 loss)
I0921 10:29:37.327343 19254 sgd_solver.cpp:106] Iteration 9600, lr =
0.00603682
I0921 10:29:37.467617 19254 solver.cpp:228] Iteration 9700, loss = 0.0730378
I0921 10:29:37.467648 19254 solver.cpp:244] Train net output #0: loss =
0.0729548 (* 1 = 0.0729548 loss)
I0921 10:29:37.467653 19254 sgd_solver.cpp:106] Iteration 9700, lr =
0.00601382
I0921 10:29:37.617055 19254 solver.cpp:228] Iteration 9800, loss = 0.0860098
I0921 10:29:37.617085 19254 solver.cpp:244] Train net output #0: loss =
0.0859269 (* 1 = 0.0859269 loss)
I0921 10:29:37.617091 19254 sgd_solver.cpp:106] Iteration 9800, lr =
0.00599102
I0921 10:29:37.747927 19254 solver.cpp:228] Iteration 9900, loss = 0.0701194
I0921 10:29:37.747957 19254 solver.cpp:244] Train net output #0: loss =
0.0700364 (* 1 = 0.0700364 loss)
I0921 10:29:37.747962 19254 sgd_solver.cpp:106] Iteration 9900, lr =
0.00596843
I0921 10:29:37.875285 19254 solver.cpp:454] Snapshotting to binaryproto file
/to/your/path/fk_iter_10000.caffemodel
I0921 10:29:37.882577 19254 sgd_solver.cpp:273] Snapshotting solver state to
binary proto file /to/your/path/fk_iter_10000.solverstate
I0921 10:29:37.887320 19254 solver.cpp:317] Iteration 10000, loss = 0.0612114
I0921 10:29:37.887341 19254 solver.cpp:337] Iteration 10000, Testing net (#0)
I0921 10:29:37.887343 19254 net.cpp:684] Ignoring source layer fk-data
I0921 10:29:38.110960 19254 solver.cpp:404] Test net output #0: loss =
0.0650334 (* 1 = 0.0650334 loss)
I0921 10:29:38.110980 19254 solver.cpp:322] Optimization Done.
I0921 10:29:38.110982 19254 caffe.cpp:222] Optimization Done.

```

16.3.4 网络二

结构图如图 16-3 所示。

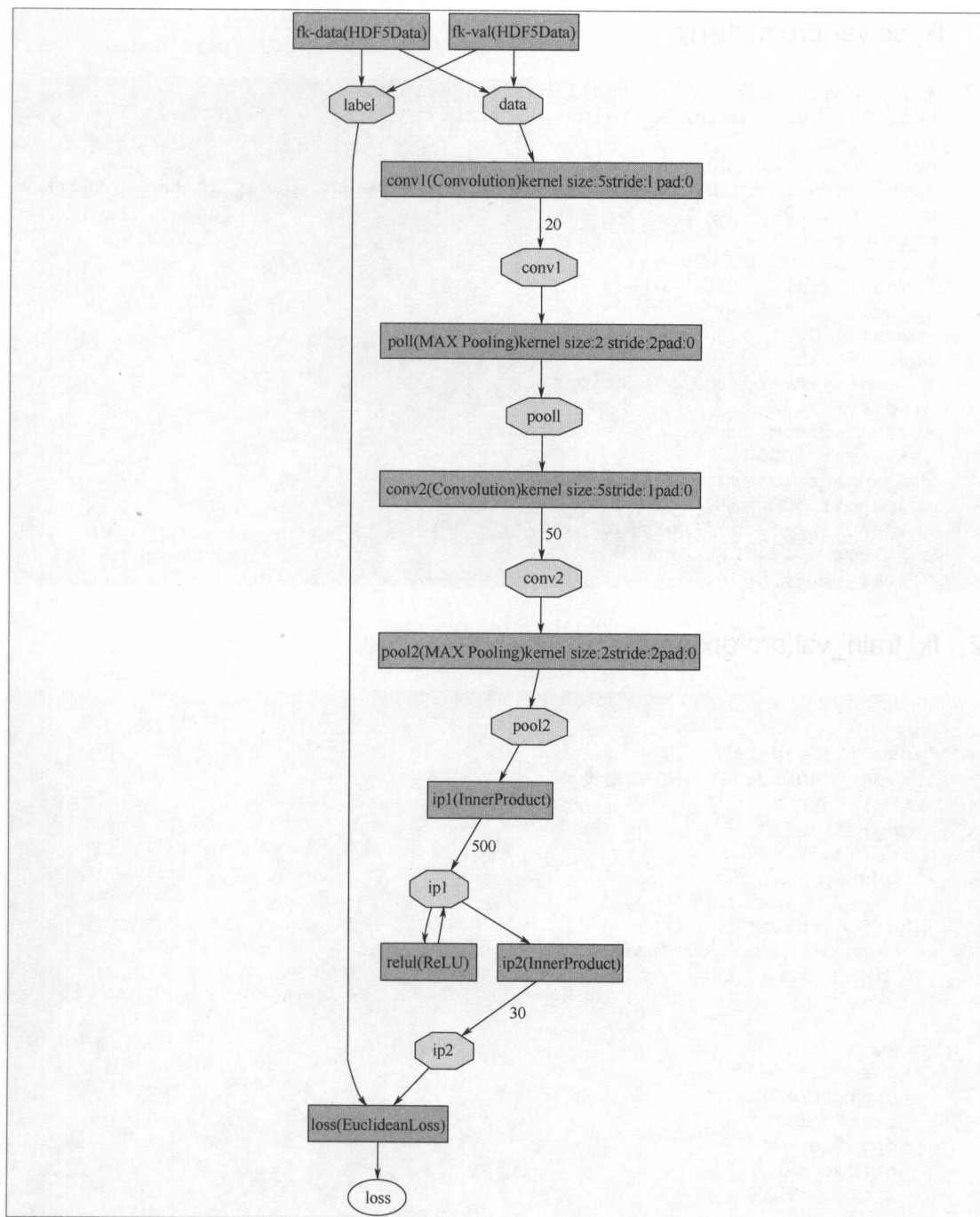


图 16-3 网络二的结构图

1. fk_solver.prototxt解读

```
# The train/test net protocol buffer definition
net: "/to/your/path/fk_train_val.prototxt"
test_iter: 1
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.001
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "fixed"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "/to/your/path/fk"
# solver mode: CPU or GPU
solver_mode: GPU
```

2. fk_train_val.prototxt解读

```
name: "FK"
layer {
  name: "fk-data"
  type: "HDF5Data" #HDF5 格式
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  hdf5_data_param {
    source: "/to/your/path/train.txt"
    batch_size: 64
  }
}
layer {
  name: "fk-val"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  hdf5_data_param {
```

```

    source: "/to/your/path/val.txt"
    batch_size: 100
  }
}

layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 100
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}

layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 30
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

```

```

    }
  }
  layer {
    name: "loss"
    type: "EuclideanLoss"
    bottom: "ip2"
    bottom: "label"
    top: "loss"
  }
}

```

3. 训练过程摘要

```

I0921 11:32:34.171994 19642 caffe.cpp:185] Using GPUs 0
I0921 11:32:34.635191 19642 caffe.cpp:190] GPU 0: GeForce GTX 1070
I0921 11:32:34.806557 19642 solver.cpp:48] Initializing solver from
parameters:
  test_iter: 100
  test_interval: 500
  base_lr: 0.001
  display: 100
  max_iter: 10000
  lr_policy: "fixed"
  gamma: 0.0001
  power: 0.75
  momentum: 0.9
  weight_decay: 0.0005
  snapshot: 5000
  snapshot_prefix: "/to/your/path/fk"
  solver_mode: GPU
  device_id: 0
  net: "/to/your/path/fk_train_val.prototxt"
I0921 11:32:34.806653 19642 solver.cpp:91] Creating training net from net
file: /to/your/path/fk_train_val.prototxt
I0921 11:32:34.806782 19642 net.cpp:313] The NetState phase (0) differed from
the phase (1) specified by a rule in layer fk-val
I0921 11:32:34.806840 19642 net.cpp:49] Initializing net from parameters:
  name: "FK"
  .....
I0921 11:35:11.069465 19642 solver.cpp:228] Iteration 9000, loss =
0.00753033
I0921 11:35:11.069478 19642 solver.cpp:244]   Train net output #0: loss =
0.00753033 (* 1 = 0.00753033 loss)
I0921 11:35:11.069483 19642 sgd_solver.cpp:106] Iteration 9000, lr = 0.001
I0921 11:35:12.650487 19642 solver.cpp:228] Iteration 9100, loss = 0.0077922
I0921 11:35:12.650516 19642 solver.cpp:244]   Train net output #0: loss =
0.00779221 (* 1 = 0.00779221 loss)
I0921 11:35:12.650519 19642 sgd_solver.cpp:106] Iteration 9100, lr = 0.001
I0921 11:35:14.231763 19642 solver.cpp:228] Iteration 9200, loss =
0.00864169
I0921 11:35:14.231783 19642 solver.cpp:244]   Train net output #0: loss =

```



```

0.00864169 (* 1 = 0.00864169 loss)
  I0921 11:35:14.231787 19642 sgd_solver.cpp:106] Iteration 9200, lr = 0.001
  I0921 11:35:15.813128 19642 solver.cpp:228] Iteration 9300, loss =
0.00642176
  I0921 11:35:15.813150 19642 solver.cpp:244] Train net output #0: loss =
0.00642176 (* 1 = 0.00642176 loss)
  I0921 11:35:15.813153 19642 sgd_solver.cpp:106] Iteration 9300, lr = 0.001
  I0921 11:35:17.394590 19642 solver.cpp:228] Iteration 9400, loss =
0.00607784
  I0921 11:35:17.394613 19642 solver.cpp:244] Train net output #0: loss =
0.00607784 (* 1 = 0.00607784 loss)
  I0921 11:35:17.394615 19642 sgd_solver.cpp:106] Iteration 9400, lr = 0.001
  I0921 11:35:18.960016 19642 solver.cpp:337] Iteration 9500, Testing net (#0)
  I0921 11:35:18.960031 19642 net.cpp:684] Ignoring source layer fk-data
  I0921 11:35:19.676095 19642 solver.cpp:404] Test net output #0: loss =
0.0243101 (* 1 = 0.0243101 loss)
  I0921 11:35:19.680619 19642 solver.cpp:228] Iteration 9500, loss =
0.00822955
  I0921 11:35:19.680631 19642 solver.cpp:244] Train net output #0: loss =
0.00822955 (* 1 = 0.00822955 loss)
  I0921 11:35:19.680635 19642 sgd_solver.cpp:106] Iteration 9500, lr = 0.001
  I0921 11:35:21.261399 19642 solver.cpp:228] Iteration 9600, loss =
0.00706134
  I0921 11:35:21.261420 19642 solver.cpp:244] Train net output #0: loss =
0.00706134 (* 1 = 0.00706134 loss)
  I0921 11:35:21.261423 19642 sgd_solver.cpp:106] Iteration 9600, lr = 0.001
  I0921 11:35:22.842291 19642 solver.cpp:228] Iteration 9700, loss = 0.0076612
  I0921 11:35:22.842314 19642 solver.cpp:244] Train net output #0: loss =
0.0076612 (* 1 = 0.0076612 loss)
  I0921 11:35:22.842319 19642 sgd_solver.cpp:106] Iteration 9700, lr = 0.001
  I0921 11:35:24.423429 19642 solver.cpp:228] Iteration 9800, loss =
0.00934706
  I0921 11:35:24.423449 19642 solver.cpp:244] Train net output #0: loss =
0.00934706 (* 1 = 0.00934706 loss)
  I0921 11:35:24.423454 19642 sgd_solver.cpp:106] Iteration 9800, lr = 0.001
  I0921 11:35:26.004557 19642 solver.cpp:228] Iteration 9900, loss =
0.00877455
  I0921 11:35:26.004580 19642 solver.cpp:244] Train net output #0: loss =
0.00877455 (* 1 = 0.00877455 loss)
  I0921 11:35:26.004582 19642 sgd_solver.cpp:106] Iteration 9900, lr = 0.001
  I0921 11:35:27.569869 19642 solver.cpp:454] Snapshotting to binary proto
file kaggle/facial-keypoints/fk_iter_10000.caffemodel
  I0921 11:35:27.676472 19642 sgd_solver.cpp:273] Snapshotting solver state
to binary proto file kaggle/facial-keypoints/fk_iter_10000.solverstate
  I0921 11:35:27.729967 19642 solver.cpp:317] Iteration 10000, loss =
0.00789725
  I0921 11:35:27.729990 19642 solver.cpp:337] Iteration 10000, Testing net
(#0)
  I0921 11:35:27.729993 19642 net.cpp:684] Ignoring source layer fk-data
  I0921 11:35:28.435431 19642 solver.cpp:404] Test net output #0: loss =
0.0228806 (* 1 = 0.0228806 loss)

```



```
I0921 11:35:28.435446 19642 solver.cpp:322] Optimization Done.
I0921 11:35:28.435448 19642 caffe.cpp:222] Optimization Done.
```

16.3.5 Python 人脸特征预测程序

运行以下 Python 程序, 我们就可以获得 test.csv 中图像的所有特征点。为了让 Python 和 Caffe 顺利结合, 我们在 fk_deploy.prototxt 中采用了 MemoryData 类型的数据层输入。

1. fk_test.py 解读

```
import numpy as np
import pandas as pd
import caffe

MODEL_FILE = './fk_deploy.prototxt'
PRETRAINED = './fk_iter_10000.caffemodel'

dataframe = pd.read_csv('/to/your/path/test.csv', header=0)
dataframe['Image'] = dataframe['Image'].apply(lambda im: np.fromstring(im,
sep=' '))
data = np.vstack (dataframe['Image'].values)
data = data.reshape([-1, 96, 96])
data = data.astype(np.float32)

# Scale between 0 and 1
data = data/255.
data = data.reshape(-1, 1, 96, 96)

net = caffe.Net(MODEL_FILE, PRETRAINED, caffe.TEST)
caffe.set_mode_gpu()

total_images = data.shape[0] #本例中, 我们把所有图片作为一个 batchsize 一次性处
理, 如果读者的内存或 GPU 显存不够大, 可以分为多个批次循环处理。
print 'Total images to be predicted:', total_images
dataL = np.zeros([total_images, 1, 1, 1], np.float32) #标签输入。在使用 Caffe 的
memorydata 类型输入时, Caffe 要求的标签 shape 必须是 [batchsize, 1, 1, 1], 事实上在本例中,
我们因为是预测, 所以并不使用这个标签, 但 Caffe 要求必须输入。

net.set_input_arrays(data.astype(np.float32), dataL.astype(np.float32))
pred = net.forward()
predicted = net.blobs['ip2'].data * 96 #转换成图像坐标

print 'Predicted', predicted

print 'Predicted shape: ', predicted.shape
print 'Saving to csv..'
```

```
np.savetxt("fkp_output.csv", predicted, delimiter=",")
```

2. 网络一的fk_deploy.prototxt解读

```
name: "FK"
layer {
  name: "fk-data"
  type: "MemoryData"
  top: "data"
  top: "label"
  memory_data_param {
    batch_size: 1783#在一次迭代中处理完所有样本，GPU 内存如不够，则此参数需要减小
    channels: 1
    width: 96
    height: 96
  }
}

layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 100
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}

layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
```

```

top: "ip2"
param {
  lr_mult: 1
}
param {
  lr_mult: 2
}
inner_product_param {
  num_output: 30
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}
}
}

```

3. 网络二的fk_deploy.prototxt解读

```

name: "FK"
layer {
  name: "fk-data"
  type: "MemoryData"
  top: "data"
  top: "label"
  memory_data_param {
    batch_size: 1783 #在一次迭代中处理完所有样本, GPU 内存如不够, 则此参数需要减小
    channels: 1
    width: 96
    height: 96
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
  }
}

```

```

    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
}

```



```

layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 30
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

```


17

第 17 章 Kaggle 项目实践： 猫狗分类检测

上一章我们利用 Caffe 框架解决了 Kaggle 数据竞赛中人脸特征检测的知识类项目，并实现了两种 Caffe 网络：一个是全连接层的网络框架，另一个是卷积层的网络框架。通过训练和测试对比，两种网络虽然都实现了人脸部特征点的检测，但后者相对前者识别精度要高很多，说明卷积层网络对人脸部特征点的刻画更准确，效果要优于全连接层网络框架。

本章我们引入 Kaggle 网站上另一个非常有趣的知识类竞赛项目：猫狗分类检测。项目的主要内容是对一组猫和狗的图片进行分类识别，达到最高分类识别成功率者获胜。

17.1 项目简介

2013 年，Kaggle 发布了第一版的猫狗分类检测项目，需要参赛者提交算法判断一张图片包含的是一只猫还是狗。这对人脑显得很容易，但对计算机判断有一定的难度。

随着最近几年机器学习，特别是深度学习和图片分析技术的发展，这一著名的猫狗分类检测项目希望能引入更新的技术得到更高的识别准确率。第二版猫狗分类检测项目在 2016 年 9 月 2 日发布至 2017 年 3 月 2 日结束，项目包含猫和狗两类动物的图片，用于训

练的图片有 25,000 张，其中猫和狗各有 12,500，图片的文件名即为图片的标签表示。用于测试的图片有 12,500 张图片，文件名以数字命名。目标是使用训练数据进行训练(Training)，预测测试图片中每一张图片是哪一种动物，用“1”表示狗，“0”表示猫。

本章将介绍在 Python 环境下，如何运用 Caffe 框架，使用深度学习来实践这个项目。以下为这个项目的链接 URL，有兴趣的读者可自行访问：

```
https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition
```

友情提醒：读者在国内注册 Kaggle 账号时，会出现无法验证通过的情况。主要原因是 Kaggle 使用了 Google 的服务，请读者自行解决。一旦账号注册成功，后续就可以正常使用。。

17.2 赛题和数据

项目提供的数据可以在 Kaggle 直接下载，以方便进行训练和测试，下载地址为：

```
https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data
```

- train.zip: 用以训练的 25,000 张带标签的猫和狗图片文件。采用 JPG 格式的图片文件，文件名以“cat.x.jpg”或“dog.y.jpg”命名，其中 x 和 y 为图片编号，文件名中的“cat”和“dog”表示了这张图片的标签。
- test.zip: 用以测试的 12,500 张测试图片。采用 JPG 格式，每张图片以数字进行编号命名，用于验证训练后算法的识别成功率。

17.3 Caffe 训练和测试数据库

17.3.1 数据库生成

由于 Caffe 自身并不支持 JPG 等图片格式，只支持 LevelDB、LMDB、HDF5 这三种格式。所以首先要将把 train.zip 解压后转换成 Caffe 能支持的数据格式。

在本项目中，项目数据为 JPG 图片格式，所以必须先把 JPG 格式转成 Caffe 支持的 LMDB 格式。格式转换的具体步骤如下：

1. 标签文件生成

Caffe 的 LMDB 数据格式需要单独输入标签文件。因此，第一步要生成训练数据对应的标签文件。首先把从 Kaggle 下载的 train.zip 解压缩，命令如下：

```
unzip train.zip
```

解压缩完成后，在当前目录下生成“train”文件夹，文件夹包含 25,000 张训练图片。接下来生成此训练数据的标签文件，生成标签的 shell 脚本文件名为“create_filelist_train.sh”，内容如下：

```
# /usr/bin/env sh
DATA=data/train
WORK=data
echo "Create train.txt..."
rm -rf $DATA/train.txt
find $DATA -name cat.*.jpg | cut -d '/' -f3 | sed "s/$/ 0/">>$DATA/train.txt
find $DATA -name dog.*.jpg | cut -d '/' -f3 | sed "s/$/ 1/">>$DATA/tmp.txt
cat $DATA/tmp.txt>>$DATA/train.txt
rm -rf $DATA/tmp.txt
mv $DATA/train.txt $WORK/
echo "Done.."
```

在 Shell 命令运行此脚本后，会在 data 目录下生成一个 train.txt 文件，打开此文件，部分内容如下：

```
cat.11682.jpg 0
cat.12413.jpg 0
cat.578.jpg 0
.....
dog.7299.jpg 1
dog.6719.jpg 1
dog.5985.jpg 1
```

实际上，此脚本的作用是把 train 文件夹的每一个图片的文件名作为字符串写入 train.txt，然后增加一个“0”或“1”的字段：“0”表示当前图片是猫，“1”表示当前图片是狗。

2. LMDB数据转换

完成 train.txt 标签文件后, 就可以用 Caffe 自带的工具 `convert_imageset` 生成 LMDB 格式的数据了, 具体实现的 Shell 脚本 `create_lmdb_train.sh` 内容如下:

```
#!/usr/bin/env sh
DATA=data/trains
WORK=data
rm -rf $DATA/catdog_train_lmdb
/home/alex/caffe-master/build/tools/convert_imageset --shuffle \
--resize_height=128 --resize_width=128 \
/home/alex/caffe-master/examples/catdog/data/train/ $WORK/train.txt
$DATA/catdog_train_lmdb
```

运行此脚本时, 请读者注意 `convert_imageset` 生成工具的路径和 `train` 图片的路径, 以及 `train.txt` 标签文件路径。其中, `convert_imageset` 的 `resize` 参数大小为 128 的目的是为了统计图片的输入大小, 过大可能会增加训练的计算量, 过小可能会影响最后的识别准确率, 请读者根据情况适当调整。完成后, 共有 250,000 个图片得到转换, Log 输出如下:

```
I0929 11:33:41.164916 6993 convert_imageset.cpp:83] Shuffling data
I0929 11:33:41.167392 6993 convert_imageset.cpp:86] A total of 25000 images.
I0929 11:33:41.167549 6993 db_lmdb.cpp:38] Opened lmdb data/train/catdog_
train_lmdb
I0929 11:33:42.363299 6993 convert_imageset.cpp:144] Processed 1000 files.
I0929 11:33:43.669893 6993 convert_imageset.cpp:144] Processed 2000 files.
I0929 11:33:44.929378 6993 convert_imageset.cpp:144] Processed 3000 files.
I0929 11:33:46.297689 6993 convert_imageset.cpp:144] Processed 4000 files.
I0929 11:33:47.632467 6993 convert_imageset.cpp:144] Processed 5000 files.
I0929 11:33:48.975553 6993 convert_imageset.cpp:144] Processed 6000 files.
I0929 11:33:50.277083 6993 convert_imageset.cpp:144] Processed 7000 files.
I0929 11:33:51.602145 6993 convert_imageset.cpp:144] Processed 8000 files.
I0929 11:33:52.927559 6993 convert_imageset.cpp:144] Processed 9000 files.
I0929 11:33:54.242007 6993 convert_imageset.cpp:144] Processed 10000 files.
I0929 11:33:55.555160 6993 convert_imageset.cpp:144] Processed 11000 files.
I0929 11:33:56.866227 6993 convert_imageset.cpp:144] Processed 12000 files.
I0929 11:33:58.192003 6993 convert_imageset.cpp:144] Processed 13000 files.
I0929 11:33:59.499935 6993 convert_imageset.cpp:144] Processed 14000 files.
I0929 11:34:00.800756 6993 convert_imageset.cpp:144] Processed 15000 files.
I0929 11:34:02.101486 6993 convert_imageset.cpp:144] Processed 16000 files.
I0929 11:34:03.397920 6993 convert_imageset.cpp:144] Processed 17000 files.
I0929 11:34:04.723562 6993 convert_imageset.cpp:144] Processed 18000 files.
I0929 11:34:06.014907 6993 convert_imageset.cpp:144] Processed 19000 files.
I0929 11:34:07.300379 6993 convert_imageset.cpp:144] Processed 20000 files.
I0929 11:34:08.590576 6993 convert_imageset.cpp:144] Processed 21000 files.
I0929 11:34:09.924612 6993 convert_imageset.cpp:144] Processed 22000 files.
I0929 11:34:11.226655 6993 convert_imageset.cpp:144] Processed 23000 files.
```

```
I0929 11:34:12.519392 6993 convert_imageset.cpp:144] Processed 24000 files.
I0929 11:34:13.762356 6993 convert_imageset.cpp:144] Processed 25000 files.
```

执行成功后, 会生成 `catdog_train_lmdb` 文件夹, 就是 Caffe 所需要的 LMDB 数据格式文件。

3. 测试数据转换

Caffe 的训练每次迭代需要进行测试验证 Loss 和 Accuracy 的值, 以便我们判断创建的 Caffe 深度学习网络及参数是否合理, 是否正常收敛, 有没有过拟合等。因此, 需要在 train 训练数据里随机抽取 10% 左右的图片进行测试验证。使用 Linux Shell 的随机变量产生 2,000 多个测试图片的脚本 `create_random.sh` 如下:

```
#!/bin/bash

for ((i=0;i<1250;++i)); do
    cp train/'cat.'$((RANDOM%12500+1)).jpg' test/
    echo cp train/'cat.'$((RANDOM%12500+1)).jpg' test/ ...
done

for ((i=0;i<1250;++i)); do
    cp train/'dog.'$((RANDOM%12500+1)).jpg' test/
    echo cp train/'dog.'$((RANDOM%12500+1)).jpg' test/ ...
done
```

执行完成后, 会在当前目录下生成 `test` 文件夹, 此文件夹内包含了随机产生的 cat 图片和 dog 图片, 共计 2,387 个图片。

接下来按照 train 训练图片的方式, 生成标签和 LMDB 格式文件, 脚本分别是 `create_filelist_test.sh` 和 `create_lmdb_test.sh`。脚本内容分别如下 (供读者参考):

`create_filelist_test.sh`

```
# /usr/bin/env sh
DATA=data/test
WORK=data
echo "Create test.txt..."
rm -rf $DATA/test.txt
find $DATA -name cat.*.jpg | cut -d '/' -f3 | sed "s/$/ 0/">>$DATA/test.txt
find $DATA -name dog.*.jpg | cut -d '/' -f3 | sed "s/$/ 1/">>$DATA/tmp.txt
cat $DATA/tmp.txt>>$DATA/test.txt
rm -rf $DATA/tmp.txt
mv $DATA/test.txt $WORK/
echo "Done.."
```

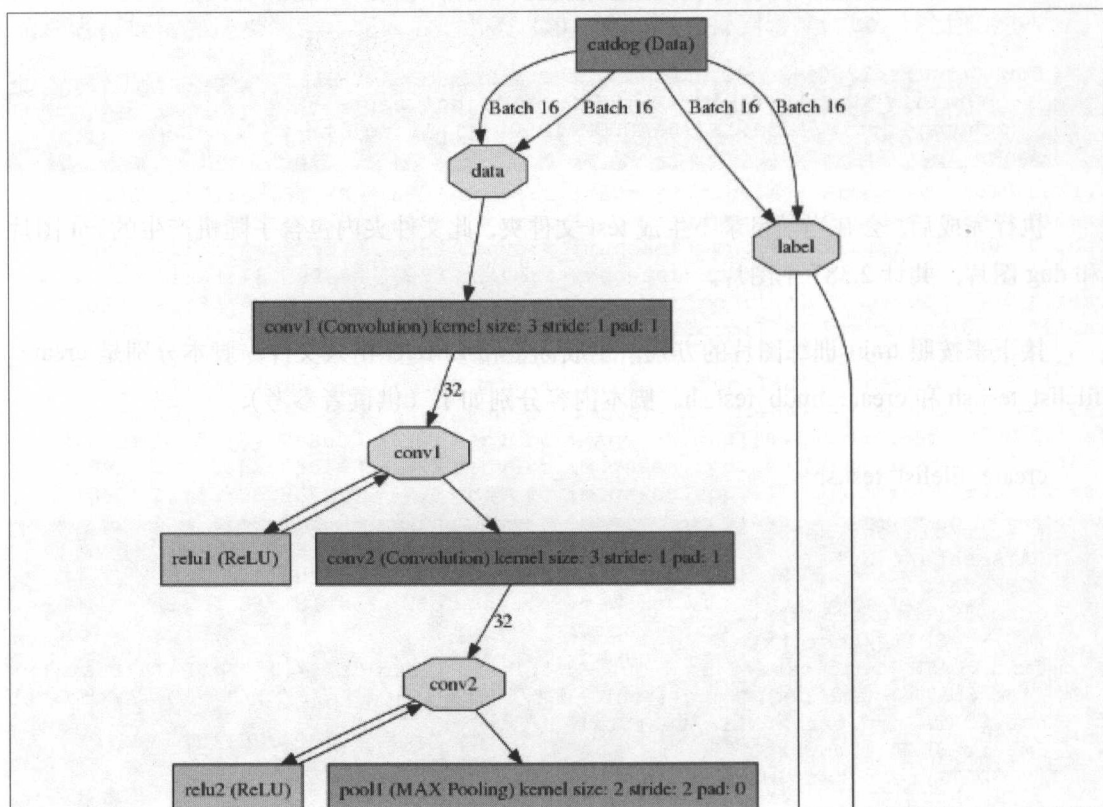


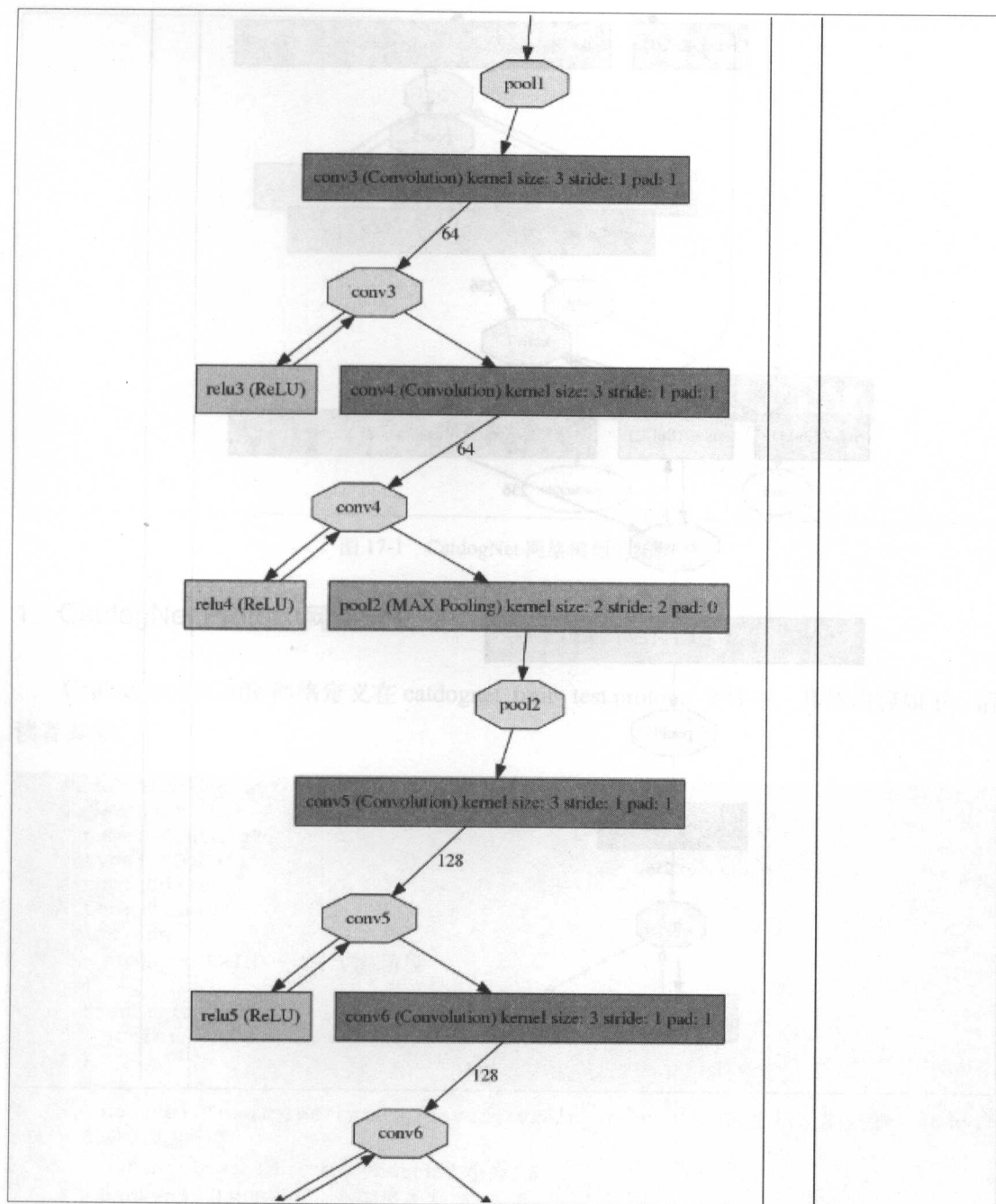
```
create_lmdb_test.sh
```

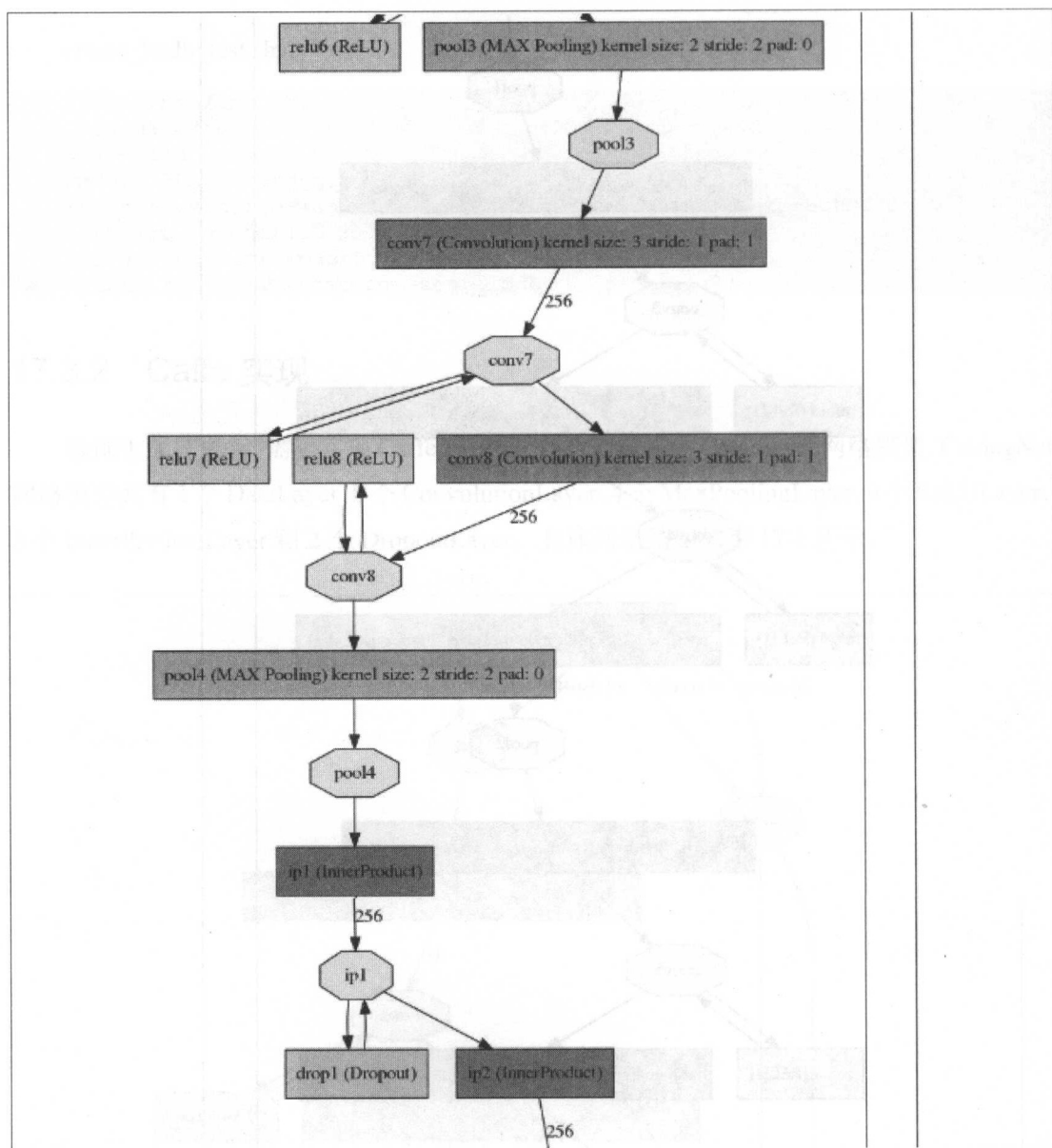
```
#!/usr/bin/env sh
DATA=data/test
WORK=data
rm -rf $DATA/catdog_test_lmdbs
/home/alex/caffe-master/build/tools/convert_imageset --shuffle \
--resize_height=128 --resize_width=128 \
/home/alex/caffe-master/examples/catdog/data/test/
$WORK/test.txt $DATA/catdog_test_lmdb
```

17.3.2 Caffe 实现

数据生成完成后, 需要实现 Caffe 训练网络, 我们取名为 CatdogNet 网络模型。CatdogNet 网络模型共有 1 个 DataLayer、8 个 ConvolutionLayer、4 个 MaxPoolingLayer、9 个 ReLULayer、3 个 InnerProductLayer 和 2 个 DropoutLayer。具体网络结构如图 17-1 所示。







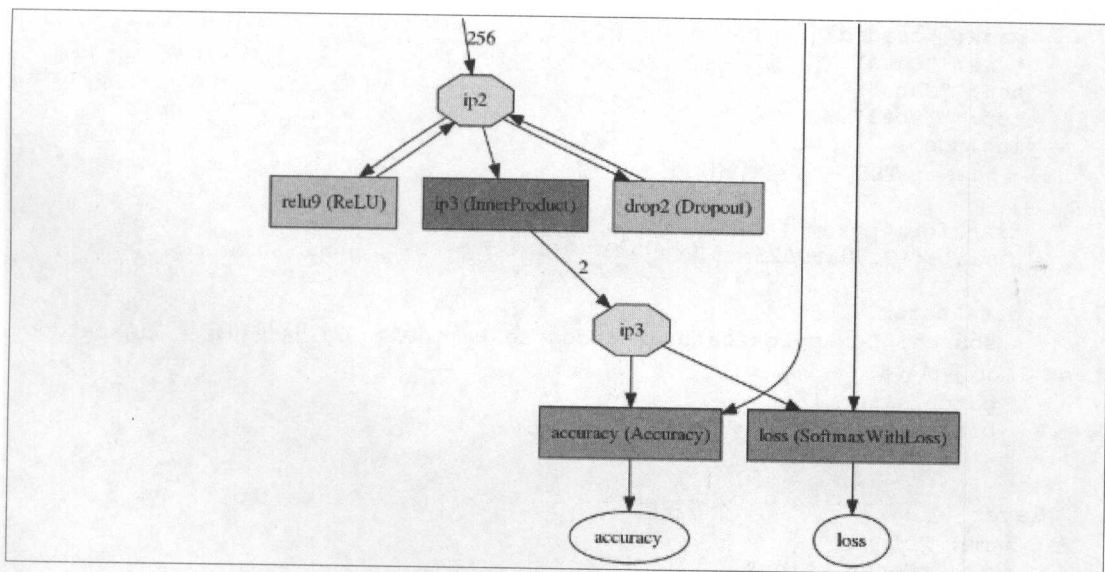


图 17-1 CatdogNet 网络模型的结构

1. CatdogNet Prototxt网络定义

CatdogNet 的 Caffe 网络定义在 catdognet_train_test.prototxt 文件中，具体内容如下，请读者参考：

```

name: "CatdogNet"
layer {
  name: "catdog"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN    // 训练阶段
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/catdog/catdog_train_lmdb" // 这里指定生成的 catdog_
    train_lmdb 训练数据
    batch_size: 16    // batch 大小为 16
    backend: LMDB    // 数据格式为 LMDB
  }
}
layer {

```

```

name: "catdog"
type: "Data"
top: "data"
top: "label"
include {
  phase: TEST // 测试阶段
}
transform_param {
  scale: 0.00390625
}
data_param {
  source: "examples/catdog/catdog_test_lmdb" // 这里指定生成的 catdog_
test_lmdb 测试数据
  batch_size: 16
  backend: LMDB
}
}
layer { // 第一个卷积层
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 32 // 输出为 32
    pad: 1 // 填充为 1, 以后每个卷积层都必须填充为 1, 以避免图片缩小;
    kernel_size: 3 // 卷积核 3*3
    stride: 1 // 步长为 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
}
layer { // 第一个 ReLU 层, 以后每一个卷积层后跟一个 ReLU 层;
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
}
layer { // 第二个卷积层
  name: "conv2"
  type: "Convolution"

```



```

    bottom: "conv1"
    top: "conv2"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {           // 参数与第一个卷积层相同;
        num_output: 32
        pad: 1
        kernel_size: 3
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "relu2"
    type: "ReLU"
    bottom: "conv2"
    top: "conv2"
}
layer {                          // 第一个 Max Pooling 层
    name: "pool1"
    type: "Pooling"
    bottom: "conv2"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 2           // 核大小 2
        stride: 2                // 步长为 2
    }
}
layer {                          // 第三个卷积层
    name: "conv3"
    type: "Convolution"
    bottom: "pool1"
    top: "conv3"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 64           // 输出为 64, 其他参数不变
    }
}

```

```

        pad: 1
        kernel_size: 3
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "relu3"
    type: "ReLU"
    bottom: "conv3"
    top: "conv3"
}
layer {                                     // 第四个卷积层
    name: "conv4"
    type: "Convolution"
    bottom: "conv3"
    top: "conv4"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 64                    // 输出为 64，其他参数不变
        pad: 1
        kernel_size: 3
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "relu4"
    type: "ReLU"
    bottom: "conv4"
    top: "conv4"
}
layer {                                     // 第二个 Max Pooling 层，参数相同
    name: "pool2"
    type: "Pooling"
    bottom: "conv4"

```

```

    top: "pool2"
    pooling_param {
      pool: MAX
      kernel_size: 2
      stride: 2
    }
  }
}
layer {                                // 第五个卷积层
  name: "conv5"
  type: "Convolution"
  bottom: "pool2"
  top: "conv5"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 128          // 输出为 128, 其他参数不变
    pad: 1
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu5"
  type: "ReLU"
  bottom: "conv5"
  top: "conv5"
}
layer {                                // 第六个卷积层
  name: "conv6"
  type: "Convolution"
  bottom: "conv5"
  top: "conv6"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 128          // 输出为 128, 其他参数不变
    pad: 1

```



```

        kernel_size: 3
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "relu6"
    type: "ReLU"
    bottom: "conv6"
    top: "conv6"
}
layer {                                // 第三个 Max Pooling 层, 参数相同
    name: "pool3"
    type: "Pooling"
    bottom: "conv6"
    top: "pool3"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
layer {                                // 第七个卷积层
    name: "conv7"
    type: "Convolution"
    bottom: "pool3"
    top: "conv7"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 256                // 输出为 256, 其他参数不变
        pad: 1
        kernel_size: 3
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
}

```

```

layer {
  name: "relu7"
  type: "ReLU"
  bottom: "conv7"
  top: "conv7"
}
layer {          // 第八个卷积层
  name: "conv8"
  type: "Convolution"
  bottom: "conv7"
  top: "conv8"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 256      // 输出为 256, 其他参数不变
    pad: 1
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu8"
  type: "ReLU"
  bottom: "conv8"
  top: "conv8"
}
layer {          // 第四个 Max Pooling 层, 参数相同
  name: "pool4"
  type: "Pooling"
  bottom: "conv8"
  top: "pool4"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}

layer {          // 第一个全连接层
  name: "ip1"
  type: "InnerProduct"

```



```

    bottom: "pool4"
    top: "ip1"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    inner_product_param {
        num_output: 256 // 输出为 256
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer { // 第一个 Dropout 层
    name: "drop1"
    type: "Dropout"
    bottom: "ip1"
    top: "ip1"
    dropout_param {
        dropout_ratio: 0.5 // 丢弃率 0.5
    }
}
layer { // 第二个全连接层, 参数相同
    name: "ip2"
    type: "InnerProduct"
    bottom: "ip1"
    top: "ip2"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    inner_product_param {
        num_output: 256
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "relu9"
    type: "ReLU"

```

```

    bottom: "ip2"
    top: "ip2"
}
layer {           // 第二个 Dropout 层
    name: "drop2"
    type: "Dropout"
    bottom: "ip2"
    top: "ip2"
    dropout_param {
        dropout_ratio: 0.5 // 丢弃率 0.5
    }
}
layer {           // 第三个全连接层
    name: "ip3"
    type: "InnerProduct"
    bottom: "ip2"
    top: "ip3"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    inner_product_param {
        num_output: 2 // 输出为 2, 因为只有猫和狗两类
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {           // Loss 层
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip3"
    bottom: "label"
    top: "loss"
}
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "ip3"
    bottom: "label"
    top: "accuracy"
    include {
        phase: TEST
    }
}
}

```

2. CatdogNet Solver定义

CatdogNet 的 solver 文件定义在 catdognet_solver.prototxt, 具体内容如下, 请读者参考:

```
# The train/test net protocol buffer definition
net: "examples/catdog/catdognet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
test_iter: 50
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.0001 //学习率为 0.0001
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "fixed" // 学习策略为固定模式
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 100000 // 10 万次迭代
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/catdog/catdognet"
# solver mode: CPU or GPU
#solver_mode: GPU
solver_mode: GPU // GPU 模式
```

17.3.3 CatdogNet 训练

定义完 CatdogNet 网络和 Solver 文件后, 就可以直接使用 Caffe train 命令训练网络。

Shell 脚本 train_catdognet.sh 文件执行后即可开始, 具体内容如下:

1. CatdogNet训练脚本

```
#!/usr/bin/env sh

./build/tools/caffe train --solver=examples/catdog/catdognet_solver.prototxt
```

2. CatdogNet训练过程

```
I0927 20:36:33.503576 2099 caffe.cpp:185] Using GPUs 0
I0927 20:36:33.966404 2099 caffe.cpp:190] GPU 0: GeForce GTX 1070
```



```

I0927 20:36:34.135030 2099 solver.cpp:48] Initializing solver from
parameters:
  test_iter: 50
  test_interval: 500
  base_lr: 0.0001
  display: 100
  max_iter: 100000
  lr_policy: "fixed"
  gamma: 0.0001
  power: 0.75
  momentum: 0.9
  weight_decay: 0.0005
  snapshot: 5000
  snapshot_prefix: "examples/catdog/catdognet"
  solver_mode: GPU
  device_id: 0
  net: "examples/catdog/catdognet_train_test.prototxt"
.....
I0927 20:36:34.374773 2099 solver.cpp:279] Solving CatDogNet
I0927 20:36:34.374783 2099 solver.cpp:280] Learning Rate Policy: fixed
I0927 20:36:34.375917 2099 solver.cpp:337] Iteration 0, Testing net (#0)
I0927 20:36:34.785334 2099 solver.cpp:404] Test net output #0: accuracy
= 0.5125
.....
I0927 20:36:46.823710 2099 solver.cpp:337] Iteration 500, Testing net (#0)
I0927 20:36:47.220813 2099 solver.cpp:404] Test net output #0: accuracy
= 0.51
.....
I0927 20:36:59.242846 2099 solver.cpp:337] Iteration 1000, Testing net (#0)
I0927 20:36:59.639879 2099 solver.cpp:404] Test net output #0: accuracy
= 0.57
.....
I0927 20:38:39.238832 2099 solver.cpp:337] Iteration 5000, Testing net (#0)
I0927 20:38:39.621924 2099 solver.cpp:404] Test net output #0: accuracy
= 0.5925
I0927 20:38:39.621948 2099 solver.cpp:404] Test net output #1: loss =
0.668159 (* 1 = 0.668159 loss)
.....
I0927 20:40:44.213394 2099 solver.cpp:337] Iteration 10000, Testing net
(#0)
I0927 20:40:44.596611 2099 solver.cpp:404] Test net output #0: accuracy
= 0.64375
I0927 20:40:44.596637 2099 solver.cpp:404] Test net output #1: loss =
0.629592 (* 1 = 0.629592 loss)
.....
I0927 20:51:11.119061 2099 solver.cpp:337] Iteration 35000, Testing net
(#0)
I0927 20:51:11.503535 2099 solver.cpp:404] Test net output #0: accuracy
= 0.8225
I0927 20:51:11.503559 2099 solver.cpp:404] Test net output #1: loss =
0.405795 (* 1 = 0.405795 loss)

```

```

.....
I0927 20:53:16.614131 2099 solver.cpp:337] Iteration 40000, Testing net
(#0)
I0927 20:53:16.998523 2099 solver.cpp:404] Test net output #0: accuracy
= 0.82875
I0927 20:53:16.998548 2099 solver.cpp:404] Test net output #1: loss =
0.37592 (* 1 = 0.37592 loss)
.....

```

经过训练,我们发现从 1 到 40,000 次迭代收敛效果不错,但 40,000 次以后精度很难进一步提升,于是修改 solver 文件,将学习率修改为 0.00001,即 base_lr 为 0.00001。然后从 40,000 次继续训练,修改训练脚本如下:

```

./build/tools/caffe train --solver=examples/catdog/catdognet_solver.prototxt \
--snapshot=examples/catdog/catdognet_iter_40000.solverstate

```

执行上述脚本后,训练的精度有一定的提升,部分 Log 输出如下:

```

I0927 21:02:54.193164 2496 caffe.cpp:185] Using GPUs 0
I0927 21:02:54.656720 2496 caffe.cpp:190] GPU 0: GeForce GTX 1070
I0927 21:02:54.831634 2496 solver.cpp:48] Initializing solver from
parameters:
test_iter: 50
test_interval: 500
base_lr: 1e-05
display: 100
max_iter: 100000
lr_policy: "fixed"
....
I0927 21:02:55.073231 2496 caffe.cpp:209] Resuming from
examples/catdog/catdognet_iter_40000.solverstate
I0927 21:02:55.089303 2496 sgd_solver.cpp:318] SGDSolver: restoring
history
I0927 21:02:55.097225 2496 caffe.cpp:219] Starting Optimization
I0927 21:02:55.097244 2496 solver.cpp:279] Solving CatDogNet
I0927 21:02:55.097246 2496 solver.cpp:280] Learning Rate Policy: fixed
I0927 21:02:55.098448 2496 solver.cpp:337] Iteration 40000, Testing net
(#0)
I0927 21:02:55.505239 2496 solver.cpp:404] Test net output #0: accuracy
= 0.8175
.....
I0927 21:25:52.664487 2496 solver.cpp:337] Iteration 95000, Testing net
(#0)
I0927 21:25:53.049620 2496 solver.cpp:404] Test net output #0: accuracy
= 0.875
I0927 21:25:53.049644 2496 solver.cpp:404] Test net output #1: loss =
0.303209 (* 1 = 0.303209 loss)
...
I0927 21:27:58.167886 2496 solver.cpp:317] Iteration 100000, loss =

```



```

0.196717
I0927 21:27:58.167906 2496 solver.cpp:337] Iteration 100000, Testing net
(#0)
I0927 21:27:58.552809 2496 solver.cpp:404] Test net output #0: accuracy
= 0.89
I0927 21:27:58.552831 2496 solver.cpp:404] Test net output #1: loss =
0.274047 (* 1 = 0.274047 loss)
I0927 21:27:58.552834 2496 solver.cpp:322] Optimization Done.
I0927 21:27:58.552837 2496 caffe.cpp:222] Optimization Done.

```

为了进一步提升准确率，将 solver 文件的 max_iter 最大迭代次数增加到 150,000 次，再次执行如下脚本：

```

./build/tools/caffe train --solver=examples/catdog/catdognet_solver2.
prototxt \
--snapshot=examples/catdog/catdognet_iter_100000.solverstate

```

执行上述脚本后，训练的精度有了进一步的提升，部分 Log 输出如下：

```

I0928 05:54:56.687441 1770 caffe.cpp:185] Using GPUs 0
I0928 05:54:57.654078 1770 caffe.cpp:190] GPU 0: GeForce GTX 1070
I0928 05:54:58.184836 1770 solver.cpp:48] Initializing solver from
parameters:
test_iter: 50
test_interval: 500
base_lr: 1e-05
display: 100
max_iter: 150000
lr_policy: "fixed"
gamma: 0.0001
power: 0.75
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "examples/catdog/catdognet"
solver_mode: GPU
device_id: 0
net: "examples/catdog/catdognet_train_test.prototxt"
....
I0928 05:54:59.149047 1770 solver.cpp:279] Solving CatDogNet
I0928 05:54:59.149052 1770 solver.cpp:280] Learning Rate Policy: fixed
I0928 05:54:59.151250 1770 solver.cpp:337] Iteration 100000, Testing net
(#0)
I0928 05:54:59.568986 1770 solver.cpp:404] Test net output #0: accuracy
= 0.87375
I0928 05:54:59.569010 1770 solver.cpp:404] Test net output #1: loss =
0.297921 (* 1 = 0.297921 loss)
....
I0928 06:13:47.213709 1770 solver.cpp:337] Iteration 145000, Testing net

```

```
(#0)
I0928 06:13:47.599159 1770 solver.cpp:404] Test net output #0: accuracy
= 0.90625
I0928 06:13:47.599181 1770 solver.cpp:404] Test net output #1: loss =
0.23338 (* 1 = 0.23338 loss)
.....
I0928 06:15:52.277304 1770 solver.cpp:337] Iteration 150000, Testing net
(#0)
I0928 06:15:52.663425 1770 solver.cpp:404] Test net output #0: accuracy
= 0.91125
I0928 06:15:52.663450 1770 solver.cpp:404] Test net output #1: loss =
0.228302 (* 1 = 0.228302 loss)
I0928 06:15:52.663453 1770 solver.cpp:322] Optimization Done.
I0928 06:15:52.663456 1770 caffe.cpp:222] Optimization Done.
```

150,000 次的迭代后, 准确率在 91.12%。当然, 笔者认为 CatdogNet 网络还可以通过一些方法进一步提升识别准确率, 比如 `resize` 大小修改为 256×256 或最后 50,000 次的学习率修改为 0.000001 等多种方式, 但这些参数的修改会加大计算量和训练时间。

进一步思考: 是否有更合适的方法, 既能减少计算量和训练时间, 又能提高识别准确率?

17.3.4 CatdogNet 模型验证

CatdogNet 训练完成以后, 就可以开始验证了。验证是将下载的 `test.zip` 解压后, 对每一图无标签的图片识别分类, Cat 为 0, Dog 为 1。笔者使用 Python 语言编写了验证程序 `catdog.py`, 具体实现如下:

```
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
import struct

caffe_root = '/home/alex/caffe-master/'
sys.path.insert(0, caffe_root + 'python')
import caffe
MODEL_FILE = '/home/alex/caffe-master/examples/catdog/catdognet.prototxt'
PRETRAINED =
'/home/alex/caffe-master/examples/catdog/catdognet_iter_150000.caffemodel'
IMAGE_FILE = '/home/alex/caffe-master/examples/catdog/data/verify/'

net = caffe.Classifier(MODEL_FILE, PRETRAINED)

for i in xrange(12500):
```

```
imgstr = IMAGE_FILE + str(i+1) + '.jpg'
#print imgstr
input_image=caffe.io.load_image(imgstr,color=True)
prediction=net.predict([input_image], oversample = False)
caffe.set_mode_cpu()
print 'predicted class:', prediction[0].argmax()
```

其中 MODEL_FILE 变量指定了 CatdogNet 的网络模型文件, 可以从 catdognet_train_test.prototxt 训练文件修改而来 (具体可以参考 LeNet 网络定义)。

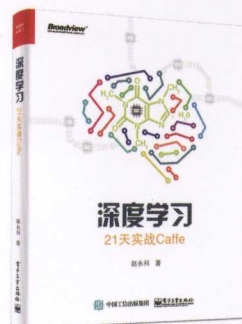
IMAGE_FILE 变量指定需要验证的数据集所在的目录, 即 test.zip 解压缩后所在的目录。

PRETRAINED 变量指定了 150000 次迭代训练的 caffemodel 文件。

执行此脚本后, 会对 test.zip 中每一张图片进行分类, 即识别图片中包含的是猫还是狗。有兴趣的读者可以将结果按竞赛项目的格式要求输出到 CSV 文件, 然后提交到 Kaggle 网站上获得最终的识别准确率。



《神经网络与深度学习》
吴岸城 著
ISBN:978-7-121-28869-2



《深度学习：21天实战Caffe》
赵永科 著
ISBN:978-7-121-29115-9

拒绝堆砌臃肿,支持纯正原创

出版事宜请关注  新浪微博 @ 半亩方塘 _
weibo.com

投稿邮箱: sxy@phei.com.cn

深度学习——Caffe之经典模型详解与实战



2016年3月, Google开发的一款人工智能程序阿尔法围棋(AlphaGo)对战世界围棋冠军、职业九段选手李世石,以4:1的总比分获胜。众多媒体和网络新闻纷纷直播或转载此次人工智能应用领域内的盛况。随后, Google在《Nature》杂志发表了关于AlphaGo算法的论文“Mastering the game of Go with deep neural networks and tree search”。此论文提到了AlphaGo用3,000万棋局训练深度神经网络的方法,展现了深度学习异常强大的学习能力。一时间,国内外掀起了研究和学习人工智能的热潮。然而,很多读者朋友希望能找到一本关于深度学习应用领域的书籍,目前市场上关于人工智能、机器学习或深度学习领域内的各类书目很多,遗憾的是这些书籍大多是理论性质的,少有系统介绍深度学习的应用实践参考书。

与此同时,笔者认为深度学习的应用能力会成为一个爆发性需求的知识技能,也会是未来科技的至高点,本书在内容上对深度学习相关的机器学习理论只做了简单介绍,更多的放在如何应用Caffe解决实际问题,并把使用当中可能出现的问题也一一列举出来,帮助读者分析原因、解决问题。本书介绍了十多种非常经典的网络模型,学习这些模型可以帮助读者很好地理解和应用Caffe框架和工具。当然,读者并无必要对这些网络模型一一阅读,也可根据自身情况选择对自己有实际帮助的案例进行分析学习。



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview

上架建议:人工智能>深度学习

ISBN 978-7-121-30118-6



9 787121 301186 >

定价: 79.00元



责任编辑: 孙学瑛
封面设计: 吴海燕